5 Sorting Algorithms

Code cùng Kiến

Mục lục

1. Quick Sort

1.1. Ý tưởng chính

- Thuật toán sắp xếp dựa trên phương pháp **"chia và trị"**.
- Thuật toán hoạt động bằng cách chọn một phần tử làm `pivot` và chia mảng thành hai phần:
  - Đoạn con bên trái bao gồm tất cả các phần tử nhỏ hơn hoặc bằng `pivot`.
  - Đoạn con bên phải bao gồm tất cả các thành phần lớn hơn `pivot`.
- Sau đó, sử dụng đệ quy để sắp xếp 2 đoạn con này. Lặp đi lặp lại cho đến khi nào cả 2 đoạn con được sắp xếp

1.2. Các bước thực hiện

- Chọn `pivot` (chốt): Có nhiều cách để chọn `pivot`, phổ biến nhất là chọn phần tử đầu tiên, phần tử cuối cùng hoặc trung bình của mảng.
- Phân chia mảng: Di chuyển các phần tử trong mảng sao cho tất cả các phần tử nhỏ hơn `pivot` nằm ở bên trái `pivot` và tất cả các phần tử lớn hơn `pivot` nằm ở bên phải `pivot`.
- Đệ quy: Gọi đệ quy cho hai mảng con bên trái và bên phải.
- Cuối cùng ghép hai mảng con đã được sắp xếp thành một mảng duy nhất.

1.3. Code tham khảo (C++)

```cpp
// Hàm phân vùng (partition)
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];  // Chọn phần tử cuối cùng làm pivot
    int i = low - 1;        // Chỉ số của phần tử nhỏ nhất

    for (int j = low; j < high; j++) {
        // Nếu phần tử hiện tại nhỏ hơn pivot
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);  // Hoán đổi arr[i] và arr[j]
        }
    }

    swap(arr[i + 1], arr[high]);  // Đưa chốt về đúng vị trí của nó
    return i + 1;  // Trả về vị trí của pivot
}

// Hàm QuickSort đệ quy
void quick_sort(vector<int>& arr, int low, int high) {
    if (low < high) {
        // Phân vùng dữ liệu
        int pivotIndex = partition(arr, low, high);

        // Sắp xếp đệ quy các phần dữ liệu bên trái và bên phải pivot
        quick_sort(arr, low, pivotIndex - 1);
        quick_sort(arr, pivotIndex + 1, high);
    }
}
```

Giải thích chi tiết:

- Hàm `partition` thực hiện phân vùng dữ liệu. Nó chọn phần tử cuối cùng làm chốt (`pivot`), sau đó sắp xếp lại các phần tử sao cho các phần tử nhỏ hơn chốt nằm bên trái và các phần tử lớn hơn chốt nằm bên phải. Cuối cùng, nó trả về vị trí của chốt.

- Hàm `quick_sort` là hàm đệ quy chính của thuật toán **Quick Sort**. Nó kiểm tra điều kiện dừng (khi `low >= high`). Nếu điều kiện không thỏa mãn, nó gọi hàm `partition` để phân vùng dữ liệu và sau đó gọi đệ quy `quick_sort` cho các phần dữ liệu bên trái và bên phải chốt.

1.4. Độ phức tạp

Tốt nhất	Trung bình	Xấu nhất
$O(n \log n)$	$O(n \log n)$	$O(n^2)$

1.5. Ưu điểm và nhược điểm

1.5.1. Ưu điểm

- Hiệu quả cao, đặc biệt với mảng lớn.
- Tốc độ sắp xếp nhanh.
- Được sử dụng trong nhiều thư viện của các ngôn ngữ như C++, Java, ...

1.5.2. Nhược điểm

- Không ổn định, có nghĩa là thứ tự ban đầu của các phần tử bằng nhau có thể bị thay đổi sau khi sắp xếp.
- Phụ thuộc vào cách chọn phần tử chốt.

1.6. Tổng kết

Quick Sort là một thuật toán sắp xếp hiệu quả và linh hoạt, được sử dụng rộng rãi trong nhiều ứng dụng khác nhau. Tuy nhiên, thuật toán này có thể không ổn định với một số trường hợp dữ liệu nhất định (nhất là khi mảng gần như được sắp xếp).

2. Radix Sort

2.1. Ý tưởng chính

- Radix Sort là một thuật toán sắp xếp không dựa trên so sánh, thay vào đó, nó sắp xếp các số bằng cách sắp xếp từng chữ số của phần tử.
- Ý tưởng chính của Radix Sort là sắp xếp các số từ chữ số ít quan trọng nhất (Least Significant Digit - LSD) đến chữ số quan trọng nhất (Most Significant Digit - MSD) hoặc ngược lại.
- Thuật toán này thường sử dụng Counting Sort hoặc Bucket Sort như một bước trung gian để sắp xếp các số theo từng chữ số.

2.2. Các bước thực hiện

- Xác định chữ số lớn nhất: Tìm chữ số có giá trị lớn nhất trong tập các số cần sắp xếp (thường là độ dài của số lớn nhất).
- Sắp xếp theo từng chữ số:
- Bắt đầu từ chữ số ít quan trọng nhất (LSD).
- Sử dụng Counting Sort (hoặc Bucket Sort) để sắp xếp các số theo chữ số hiện tại.
- Di chuyển lên chữ số tiếp theo và lặp lại quá trình cho đến khi tất cả các chữ số được sắp xếp.
- Giả sử ta có một mảng các số nguyên dương. Quy trình cụ thể như sau:
  - Bước 1: Tìm số lớn nhất trong mảng để xác định số chữ số lớn nhất (gọi là d).
  - Bước 2: Sắp xếp mảng theo từng chữ số từ LSD đến MSD.
- Với mỗi chữ số i từ 1 đến d: Sử dụng Counting Sort để sắp xếp các phần tử theo chữ số i.

2.3. Code tham khảo (C++)

```cpp
void radixSort(vector<int>& a) {
    int maxNumber = *max_element(a.begin(), a.end());
    int exp = 1; // Khởi tạo giá trị exp = 1 (để bắt đầu từ chữ số đơn vị)
    vector<int> res(a.size());
    int n = a.size();

    while (maxNumber / exp > 0) { // Lặp lại cho đến khi xử lý hết các chữ số
        vector<int> dem(10, 0);
        // Đếm số lượng số có chữ số tương ứng
        for (int i = 0; i < n; i++) dem[(a[i] / exp) % 10]++;
        for (int i = 1; i < 10; i++) dem[i] += dem[i - 1];

        for (int i = n - 1; i >= 0; i--) {
            res[dem[(a[i] / exp) % 10] - 1] = a[i];
            dem[(a[i] / exp) % 10]--;
        }

        for (int i = 0; i < n; i++) a[i] = res[i];
        exp *= 10;
    }
}
```

2.4. Độ phức tạp

Tốt nhất	Trung bình	Xấu nhất
$O(n * k)$	$O(n * k)$	$O(n * k)$

Trong đó:

- n là số lượng phần tử trong mảng.
- k là số chữ số của số lớn nhất.

Radix Sort hoạt động hiệu quả khi k là một hằng số nhỏ so với n, điều này làm cho độ phức tạp của nó gần như là $O(n)$.

2.5. Ưu điểm và nhược điểm

2.5.1. Ưu điểm:

- Radix sort cài đặt thuận tiện với các mảng với khóa sắp xếp là chuỗi (ký tự và số) hơn là khóa số do tránh được chi phí lấy các chữ số của từng số.
- Radix sort thích hợp cho sắp xếp trên **danh sách liên kết**.

2.5.2. Nhược điểm:

- Khi sắp xếp với dãy không nhiều phần tử, thuật toán này sẽ mất ưu thế so với các thuật toán khác.
- Không dùng để sắp xếp các mảng có số nguyên âm.

2.6. Tổng kết

- Radix Sort là một thuật toán sắp xếp **không dựa trên so sánh**, có thể đạt được độ phức tạp gần tuyến tính trong các trường hợp cụ thể.
- Nó hoạt động hiệu quả đối với các tập dữ liệu có kích thước vừa phải và có giá trị các chữ số đồng nhất.
- Tuy nhiên, Radix Sort yêu cầu không gian bộ nhớ bổ sung và không phải là lựa chọn tốt nhất cho tất cả các loại dữ liệu, đặc biệt là khi các số chữ số rất lớn.
- So với các thuật toán sắp xếp dựa trên so sánh như Quick Sort hay Merge Sort, Radix Sort có thể nhanh hơn trong một số trường hợp nhất định nhưng cũng có hạn chế về ứng dụng thực tế.

3. Merge Sort

3.1. Ý tưởng chính

- Giống như Quick sort, Merge sort là một thuật toán **chia để trị**. Thuật toán này chia mảng cần sắp xếp thành hai nửa.
- Sau đó lặp lại việc chia mảng ở các nửa mảng đã chia. Cuối cùng gộp các nửa đó thành mảng đã sắp xếp.

3.2. Các bước thực hiện

- Chia đôi **(Divide)**: Chia đôi dữ liệu đầu vào thành hai phần bằng nhau cho đến khi mỗi phần chỉ còn chứa **một phần tử**.
- Trộn **(Merge)**: Trộn hai phần đã được sắp xếp thành một phần mới đã được sắp xếp. Lặp lại quá trình trộn cho đến khi toàn bộ dữ liệu được sắp xếp.

3.3. Code tham khảo (C++)

```cpp
// Hàm trộn hai mảng con đã sắp xếp
void merge(vector<int>& arr, int left, int mid, int right) {
    int leftSize = mid - left + 1;
    int rightSize = right - mid;

    // Tạo hai mảng tạm để lưu trữ các nửa
    vector<int> leftArr(leftSize);
    vector<int> rightArr(rightSize);

    // Sao chép dữ liệu vào hai mảng tạm
    for (int i = 0; i < leftSize; i++)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < rightSize; j++)
        rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;  // Các chỉ số của mảng con và mảng đầu vào

    // Trộn hai mảng con vào mảng đầu vào
    while (i < leftSize && j < rightSize) {
        if (leftArr[i] <= rightArr[j])
            arr[k++] = leftArr[i++];
        else
            arr[k++] = rightArr[j++];
    }

    // Sao chép các phần tử còn lại của mảng con (nếu có)
    while (i < leftSize)
        arr[k++] = leftArr[i++];
    while (j < rightSize)
        arr[k++] = rightArr[j++];
}

// Hàm MergeSort đệ quy
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;  // Tính điểm giữa

        // Sắp xếp đệ quy hai nửa
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
```

```
        // Trộn hai nửa đã sắp xếp
        merge(arr, left, mid, right);
    }
}
```

3.4. Độ phức tạp

Tốt nhất	Trung bình	Xấu nhất
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

3.5. Ưu điểm và nhược điểm

3.5.1. Ưu điểm:

- Có độ phức tạp $O(n \log n)$ trong mọi trường hợp,điều này làm cho Merge Sort tỏ ra khá hiệu quả.
- Merge sort là một lựa chọn khi cần một thuật toán để sắp xếp có tính ổn định, khác với quick sort (một thuật toán không ổn định cho lắm:)).

3.5.2. Nhược điểm:

Không hiệu quả về mặt không gian. khi so sánh với Quick Sort về độ phức tạp không gian, của Merge Sort là $O(n)$ trong khi đó của Quick Sort chỉ là $O(1)$.

3.6. Tổng kết

Merge Sort là một thuật toán sắp xếp mạnh mẽ với độ phức tạp thời gian ổn định $O(n \log n)$. Tuy nhiên, nó đòi hỏi thêm không gian bộ nhớ phụ và không phải là thuật toán sắp xếp tại chỗ, điều này có thể hạn chế việc sử dụng trong các hệ thống có bộ nhớ hạn chế. Tuy nhiên, với các ứng dụng cần sắp xếp chính xác và hiệu quả trên các tập dữ liệu lớn, Merge Sort vẫn là một lựa chọn tuyệt vời.

4. Tim Sort

4.1. Ý tưởng

- Tim Sort là một thuật toán sắp xếp khá hiện đại, được thiết kế bởi Tim Peters vào năm 2002
- Thuật toán này kết hợp hai kỹ thuật sắp xếp khác nhau: Merge Sort và Insertion Sort, nhằm đạt được hiệu quả cao trong việc sắp xếp dữ liệu khi tận dụng tốt khả năng sắp xếp trên đoạn nhỏ của Insertion Sort và tính ổn định khi sắp xếp trên đoạn lớn của Merge Sort.

4.2. Các bước thực hiện

- Tim Sort hoạt động dựa trên việc chia nhỏ dữ liệu đầu vào thành các đoạn (runs) có kích thước nhỏ, sau đó sử dụng Insertion Sort để sắp xếp các đoạn này.
- Sau khi các đoạn đã được sắp xếp, Tim Sort sẽ sử dụng Merge Sort để hợp nhất các đoạn này thành một mảng sắp xếp hoàn chỉnh.
- Quá trình này được thực hiện như sau:
  - Chia mảng thành các đoạn con (RUN): Tim Sort đầu tiên sẽ chia mảng thành các đoạn con có số lượng phần tử nhỏ, thường là 32 hoặc 64 phần tử.
  - Các đoạn này sẽ được sắp xếp bằng Insertion Sort vì nó hoạt động rất hiệu quả với các đoạn con có số lượng phần tử nhỏ và dữ liệu gần như đã sắp xếp.
  - Hợp nhất các đoạn bằng Merge Sort: Sau khi các đoạn đã được sắp xếp, Tim Sort sẽ sử dụng Merge Sort để hợp nhất các đoạn này lại với nhau.
  - Merge Sort có độ phức tạp O(n log n) trong mọi trường hợp, giúp quá trình diễn ra một cách nhanh nhất có thể.

4.3. Code tham khảo (C++)

```cpp
// Chọn giá trị RUN
const int RUN = 32;

// Hàm sắp xếp bằng InsertionSort
void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int temp = arr[i];
        int j = i - 1;

        while (j >= left && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = temp;
    }
}

// Hàm trộn hai mảng con đã sắp xếp
void merge(vector<int>& arr, int left, int mid, int right) {
  // ... code tương tự hàm merge của Merge Sort
}

// Hàm TimSort
void timSort(vector<int>& arr, int n) {
    for (int i = 0; i < n; i += RUN)
        insertionSort(arr, i, min((i + RUN - 1), (n - 1)));
```

```
    for (int size = RUN; size < n; size = 2 * size) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = left + size - 1;
            int right = min((left + 2 * size - 1), (n - 1));

            if (mid < right)
                merge(arr, left, mid, right);
        }
    }
}
```

4.4. Độ phức tạp:

Tốt nhất	Trung bình	Xấu nhất
$O(n)$	$O(n \log n)$	$O(n \log n)$

4.5. Ưu điểm và nhược điểm

4.5.1. Ưu điểm

- Hiệu suất tốt với dữ liệu thực tế: Tim Sort làm việc tốt với nhiều loại dữ liệu, đặc biệt là những dữ liệu có tính chất gần như đã sắp xếp, rất phổ biến.
- Tận dụng lợi thế của cả Insertion Sort và Merge Sort: Tim Sort có thể tận dụng lợi thế của cả hai phương pháp, giúp cải thiện hiệu suất của cả ba trường hợp.

4.5.2. Nhược điểm

- Sử dụng bộ nhớ không hiệu quả: Tim Sort sử dụng nhiều bộ nhớ hơn so với nhiều thuật toán khác.
- Khó triển khai: Timsort phức tạp hơn so với một số thuật toán sắp xếp cơ bản khác. Điều này có thể làm cho việc triển khai và sửa lỗi trở nên khó khăn hơn, đặc biệt đối với các newbie.
- Hiệu suất không ổn định với các trường hợp đặc biệt: Trong trường hợp dữ liệu hoàn toàn ngẫu nhiên và không có bất kỳ sự sắp xếp sẵn nào, Tim Sort có thể không nhanh hơn so với các thuật toán sắp xếp khác như Quick Sort hoặc Merge Sort.
- Cần điều chỉnh tham số: Hiệu suất của Tim Sort phụ thuộc vào việc lựa chọn giá trị tham số RUN (ngưỡng để sử dụng InsertionSort). Nếu chọn giá trị không phù hợp, hiệu suất có thể bị ảnh hưởng.

4.6. Tổng kết

- Tim Sort là một thuật toán sắp xếp hiện đại và hiệu quả, kết hợp các ưu điểm của Insertion Sort và Merge Sort để đạt được hiệu suất cao.
- Với khả năng xử lý tốt các bộ dữ liệu lớn và phức tạp, Tim Sort đã trở thành một phần không thể thiếu trong nhiều ngôn ngữ lập trình ngày nay, góp phần vào việc tối ưu hóa hiệu suất của các ứng dụng phần mềm.

5. Intro Sort

5.1. Ý tưởng chính

Intro Sort bắt đầu bằng cách sử dụng Insertion Sort cho một phần đầu của dãy số. Nếu phần còn lại của dãy quá lớn, nó sẽ chuyển sang sử dụng Heap Sort. Nếu phần còn lại vẫn quá lớn, nó sẽ chuyển sang sử dụng Quick Sort, nhưng với một cơ chế để ngăn ngừa trường hợp xấu nhất của Quick Sort. (Tim Sort thì kết hợp 2 còn Intro thì ông cố nội hơn :))) lụm 3 thuật toán kết hợp lại :DD)

5.2. Các bước thực hiện

- **Bước 1:** Nếu dãy có độ dài nhỏ hơn hoặc bằng 16, Intro Sort sẽ sử dụng Insertion Sort để sắp xếp toàn bộ dãy.
- **Bước 2:** Nếu dãy có độ dài lớn hơn 16, Intro Sort sẽ chia dãy thành hai phần: phần đầu có 16 phần tử và phần còn lại.
  - Phần đầu 16 phần tử được sắp xếp bằng Insertion Sort.
  - Nếu phần còn lại có độ dài nhỏ hơn hoặc bằng 16, Intro Sort sẽ sử dụng Insertion Sort để sắp xếp phần này.
  - Nếu phần còn lại có độ dài lớn hơn 16 nhưng nhỏ hơn một ngưỡng nhất định (thường là $2 * \log(n)$, với n là độ dài của mảng), Intro Sort sẽ sử dụng Heap Sort.
  - Nếu phần còn lại có độ dài lớn hơn ngưỡng đó, Intro Sort sẽ sử dụng Quick Sort với một số điều kiện để tránh trường hợp xấu nhất.
- **Bước 3:** Sau khi sắp xếp hai phần, Intro Sort sẽ trộn chúng lại thành một dãy được sắp xếp hoàn chỉnh.

5.3. Code tham khảo (C++)

```
// Hàm sắp xếp nửa đầu mảng
void insertionSort(int arr[], int left=0, int right=16) {
  // ... code Insertion Sort
}

// Hàm sắp xếp đoạn lớn hơn 16 phần tử bằng Heap Sort
void heapSort(int arr[], int left, int right) {
    // ... code Heap Sort
}

// Hàm phân vùng cho Quick Sort
int partition(int arr[], int left, int right) {
    // ... tương tự như hàm partition của Quick Sort
}

// Hàm Quick Sort với cơ chế tránh trường hợp xấu nhất
void introSort(int arr[], int left, int right) {
    int maxDepth = 2 * log(right - left);

    // Sắp xếp nửa đầu mảng bằng Insertion Sort
    insertionSort(arr, left, min(left + 15, right));

    // Sử dụng Insertion Sort nếu đoạn nhỏ
    if (right - left <= 16) {
        insertionSort(arr, left, right);
        return;
    }
```

```
    // Sử dụng Heap Sort nếu đoạn vừa
    if (right - left > 16 && right - left <= 2 * 32) {
        heapSort(arr, left, right);
        return;
    }

    // Sử dụng Quick Sort với cơ chế tránh trường hợp xấu nhất
    if (maxDepth == 0) {
        heapSort(arr, left, right);
    } else {
        int pivot = partition(arr, left, right);
        introSort(arr, left, pivot);
        introSort(arr, pivot + 1, right);
    }
}
```

5.4. Độ phức tạp

Tốt nhất	Trung bình	Xấu nhất
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

5.5. Ưu điểm và nhược điểm

5.5.1. Ưu điểm

- Hiệu năng tốt trong mọi trường hợp: Intro Sort kết hợp các ưu điểm của Insertion Sort, Heap Sort và Quick Sort để đảm bảo hiệu năng tốt trong mọi trường hợp, cả trường hợp tốt nhất lẫn trường hợp xấu nhất.
- Tránh được trường hợp xấu nhất của Quick Sort: Intro Sort sử dụng một cơ chế để giới hạn độ sâu đệ quy của Quick Sort, từ đó tránh được trường hợp xấu nhất khi dữ liệu đã được sắp xếp hoặc đảo ngược.
- Hiệu quả với các dãy nhỏ: Intro Sort sử dụng Insertion Sort cho các dãy nhỏ, đây là giải thuật hiệu quả hơn so với Heap Sort và Quick Sort đối với các dãy nhỏ.

5.5.2. Nhược điểm

- Phức tạp hơn các giải thuật sắp xếp cơ bản: Intro Sort là một giải thuật phức tạp hơn so với các giải thuật sắp xếp cơ bản như Insertion Sort, Merge Sort hoặc Quick Sort. Điều này có thể khiến việc hiểu và triển khai nó trở nên khó khăn hơn.
- Sử dụng bộ nhớ nhiều hơn: Intro Sort sử dụng bộ nhớ nhiều hơn so với một số giải thuật sắp xếp khác, đặc biệt là khi sử dụng Heap Sort cho các dãy vừa phải.
- Một số trường hợp cụ thể có thể kém hiệu quả: Trong một số trường hợp cụ thể, Intro Sort có thể kém hiệu quả hơn so với các giải thuật sắp xếp chuyên biệt khác được tối ưu cho trường hợp đó.

5.6. Tổng kết

- Intro Sort luôn có độ phức tạp O(n log n) trong mọi trường hợp, điều này làm cho nó trở thành một giải thuật sắp xếp hiệu quả và ổn định.
- Intro Sort được sử dụng rộng rãi trong các thư viện sắp xếp tiêu chuẩn như C++ STL, Java Collections Framework và nhiều nơi khác. Nó kết hợp hiệu quả của các giải thuật sắp xếp khác nhau để trở thành một giải thuật sắp xếp đáng tin cậy và hiệu quả trong mọi trường hợp.