Báo cáo thực nghiệm

I. Mở đầu

Lí do cần phải đánh giá các thuật toán sắp xếp:

- Mỗi thuật toán có cách hoạt động khác nhau. Vì thế, chúng sẽ chạy tốt trong các trường hợp lí tưởng và ngược lại
- Tùy vào nhu cầu sử dụng, các thuật toán sắp xếp sẽ có những ưu điểm và nhược điểm khác nhau. Vì vậy, ta cần phải đánh giá chúng để phân tích và đưa ra sự lựa chọn phù hợp

*Thuật toán sắp xếp được đánh giá là ổn định khi các phần tử bằng với giá trị bằng nhau sẽ giữ nguyên thứ tự trong mảng trước khi sắp xếp. Ngược lại, các phần tử có giá trị bằng nhau với các thứ tự bất kỳ sẽ khiến thuật toán đó trở nên không ổn định.

Khi so sánh giữa các thuật toán thường sẽ có các vấn đề sau đây:

- Thời gian chạy. Đối với các dữ liệu rất lớn, các thuật toán không hiệu quả sẽ chạy rất chậm và không thể ứng dụng trong thực tế.
- Bộ nhớ. Các thuật toán nhanh đòi hỏi đệ quy sẽ có thể phải tạo ra các bản copy của dữ liệu đang xử lí. Với những hệ thống mà bộ nhớ có giới hạn (ví dụ như hệ thống nhúng), một vài thuật toán sẽ không thể chạy được.
- Độ ổn định. Độ ổn định được định nghĩa dựa trên điều gì sẽ xảy ra với các phần tử có giá trị giống nhau.

II. So sánh các thuật toán sắp xếp

1. Quicksort

1.1 Mô tả

- Đây là một thuật toán dạng “Chia để trị”, hoạt động bằng cách chọn một phần tử trong mảng đầu làm chốt “pivot” trong mảng. Sau đó phân hoạch các phần tử khác thành 2 mảng con, một mảng con bao gồm các phần tử nhỏ hơn hoặc bằng pivot, và phần còn lại là các phần tử lớn hơn pivot.
- Sau đó, thuật toán sử dụng đệ quy trên hai mảng con này để sắp xếp chúng.

1.2 Độ phức tạp

- Tốt nhất: $n\log n$
- Trung bình: $n\log n$
- Xấu nhất: n^2

1.3 Ưu và nhược điểm

- Quicksort chạy rất nhanh và dễ cài đặt, nên thường được sử dụng trong các thư viện của Java, C++,... và được sử dụng rộng rãi trên toàn thế giới
- Trong trường hợp xấu nhất, độ phức tạp của Quicksort là n^2. Nếu pivot không tốt có thể dẫn đến lỗi. Ngoài ra, Quicksort không ổn định khi chạy trên những tập dữ liệu có cấu trúc phức tạp

1.4 Cài đặt

Lưu ý: Chọn pivot là phần tử nằm giữa trong mảng

```
int partition(double arr[], int low, int high) {
    double pivot = arr[(low + high) / 2];
    int i = low - 1;
    int j = high + 1;

    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i >= j) {
            return j;
        }
        swap(arr[i], arr[j]);
    }
}

void quicksort(double arr[], int low, int high) {
    if (low < high) {
        int p = partition(arr, low, high);
        quicksort(arr, low, p);
        quicksort(arr, p + 1, high);
    }
}
```

2. Mergesort

1.1 Mô tả

- Mergesort là một thuật toán sắp xếp đệ quy, chia một danh sách thành hai phần, sắp xếp các phần đó riêng biệt và sau đó trộn lại chúng để tạo ra một danh sách được sắp xếp hoàn chỉnh.

1.2 Độ phức tạp

- Tốt nhất: nlogn
- Trung bình: nlogn
- Xấu nhất: nlogn

1.3 Ưu và nhược điểm

- Mergesort là một trong những thuật toán sắp xếp nhanh nhất và mang tính ổn định cao, thích hợp để sắp xếp các danh sách có kích thước lớn.
- Cài đặt phức tạp, thường chậm hơn các thuật toán khác khi sắp xếp các danh sách nhỏ vì phải tốn thời gian trộn các danh sách con lại với nhau. Ngoài ra, mergesort cần nhiều bộ nhớ để lưu trữ các mảng con

1.4 Cài đặt

```
void merge(double array[], int const l, int const m, int const r){
    // Tạo các mảng con left và right
        auto const sArr1 = m - l + 1;
        auto const sArr2 = r - m;
        auto *lArr = new int[sArr1], *rArr = new int[sArr2];
        for (auto i = 0; i < sArr1; i++)
                lArr[i] = array[l + i];
        for (auto j = 0; j < sArr2; j++)
                rArr[j] = array[m + 1 + j];

        auto index_1 = 0,
                 index_2 = 0;
        int index_m = l;

        // Merge các temp arrays lại với nhau
        while (index_1 < sArr1 && index_2 < sArr2) {
                if (lArr[index_1] <= rArr[index_2]) {
                        array[index_m] = lArr[index_1];
                        index_1++;
                } else {
                        array[index_m] = rArr[index_2];
                        index_2++;
                }
                index_m++;
        }

        while (index_1 < sArr1) {
                array[index_m] = lArr[index_1];
                index_1++;
                index_m++;
        }

        while (index_2 < sArr2) {
                array[index_m] = rArr[index_2];
                index_2++;
                index_m++;
        }
        delete[] lArr;
        delete[] rArr;
}

void mergeSort(double array[], int const begin, int const end)
{
        if (begin >= end) return;
        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
```

```
        merge(array, begin, mid, end);
}
```

3. Heapsort

1.1 Mô tả 1.2 Độ phức tạp

- Tốt nhất: nlogn
- Trung bình: nlogn
- Xấu nhất: nlogn

1.3 Ưu và nhược điểm

- Heapsort chạy nhanh, đặc biệt là với các mảng đã sắp xếp một phần hoặc với số lượng đầu vào ít. Ngoài ra, Heapsort sử dụng ít bộ nhớ vì không yêu cầu thêm bộ nhớ khi sử dụng
- Không ổn định và thiếu sự linh hoạt.

1.4 Cài đặt

```
void heapify(double arr[], int N, int i){
        int largest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;
        if (l < N && arr[l] > arr[largest])
                largest = l;
        if (r < N && arr[r] > arr[largest])
                largest = r;
        if (largest != i) {
                swap(arr[i], arr[largest]);
                heapify(arr, N, largest);
        }
}
void heapSort(double arr[], int N)
{
        for (int i = N / 2 - 1; i >= 0; i--)
                heapify(arr, N, i);
        for (int i = N - 1; i > 0; i--) {
                swap(arr[0], arr[i]);
                heapify(arr, i, 0);
        }
}
```

4. C++ Sort

1.1 Mô tả

C++ Sort sử dụng thuật toán Introsort (sự kết hợp giữa Quicksort, Heapsort và Insertion Sort), nhờ đó tăng độ hiệu quả khi sắp xếp.

1.2 Độ phức tạp

- Tốt nhất: nlogn
- Trung bình: nlogn
- Xấu nhất: nlogn

1.3 Ưu và nhược điểm

- std::sort() giải quyết được những nhược điểm của các thuật toán sort khác. Vì vậy, C++ Sort chạy rất nhanh chóng

1.4 Cài đặt

```
int arr[];
int n = sizeof(arr) / sizeof(arr[0]);
sort(arr, arr + n);
```

III. Kết quả thực nghiệm

Một số lỗi trong quá trình code cũng như chạy thử:

- Khi chạy thuật toán *Quicksort* với pivot ở phần tử cuối cùng trong mảng -> chạy data1.txt với dãy số vốn đã sắp xếp tăng dần thì xảy ra lỗi chạy quá thời gian (tương tự với pivot = arr[0] trong data2.txt).

Nguyên nhân: thuật toán sẽ duyệt n-1 lần trong mảng chứa 1 triệu phần tử vì không thỏa điều kiện a[i] < a[pivot].

Cách giải quyết: Chọn pivot là phần tử trung vị hoặc random bất kì vị trí nào trong mảng cần sắp xếp

Thông tin khảo sát

- Mỗi bộ data chứa một triệu số thực. Trong đó, data1 đã được sắp xếp tăng dần, bộ data2 sắp xếp giảm dần, còn 8 bộ data còn lại theo thứ tự ngẫu nhiên

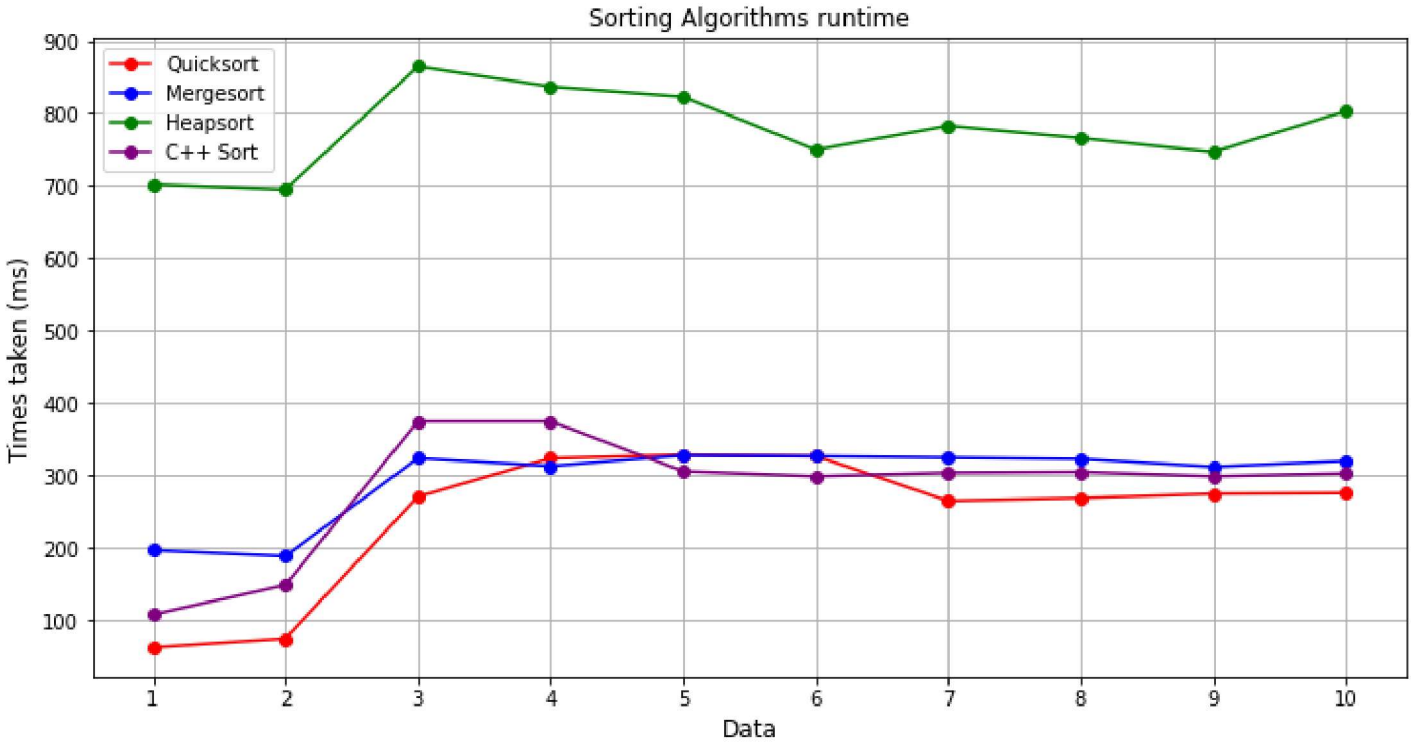
- Khảo sát được tiến hành trên Google Colab. Ngoài ra, kết quả khảo sát sẽ cho ra kết quả khác nhau nếu sử dụng trên những CPU khác nhau.

Thời gian chạy của các thuật toán trong một lần test:

Có thể thử nghiệm trên notebook trên Github

Dataset	Quicksort	Mergesort	Heapsort	C++ Sort
1	63ms	197ms	702ms	108ms
2	75ms	189ms	695ms	149ms
3	271ms	324ms	865ms	375ms
4	324ms	313ms	837ms	375ms
5	329ms	328ms	823ms	306ms
6	327ms	327ms	750ms	299ms
7	265ms	325ms	783ms	304ms
8	269ms	323ms	766ms	305ms
9	275ms	312ms	747ms	299ms
10	276ms	320ms	803ms	303ms

Biểu đồ

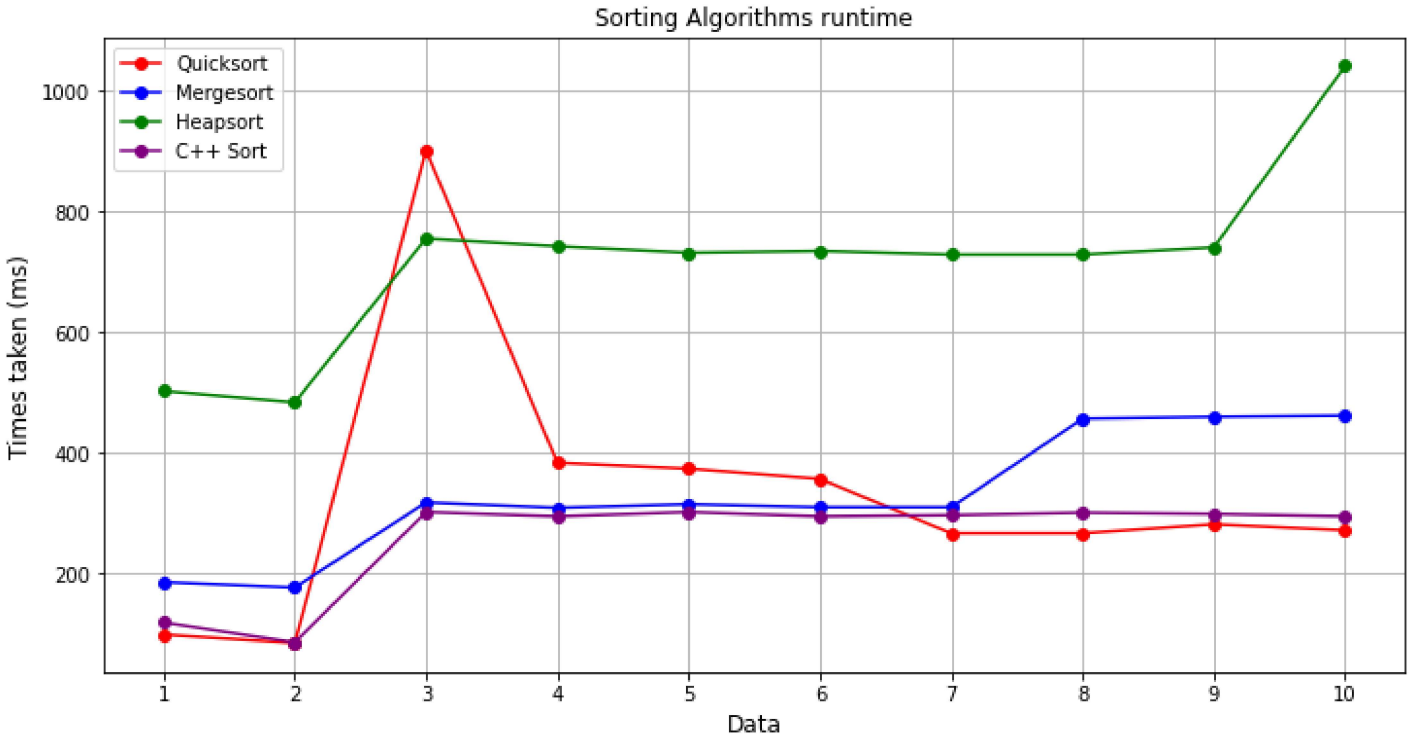


Thời gian chạy trung bình

Quicksort	Mergesort	Heapsort	C++ Sort
250ms	295ms	780ms	280ms

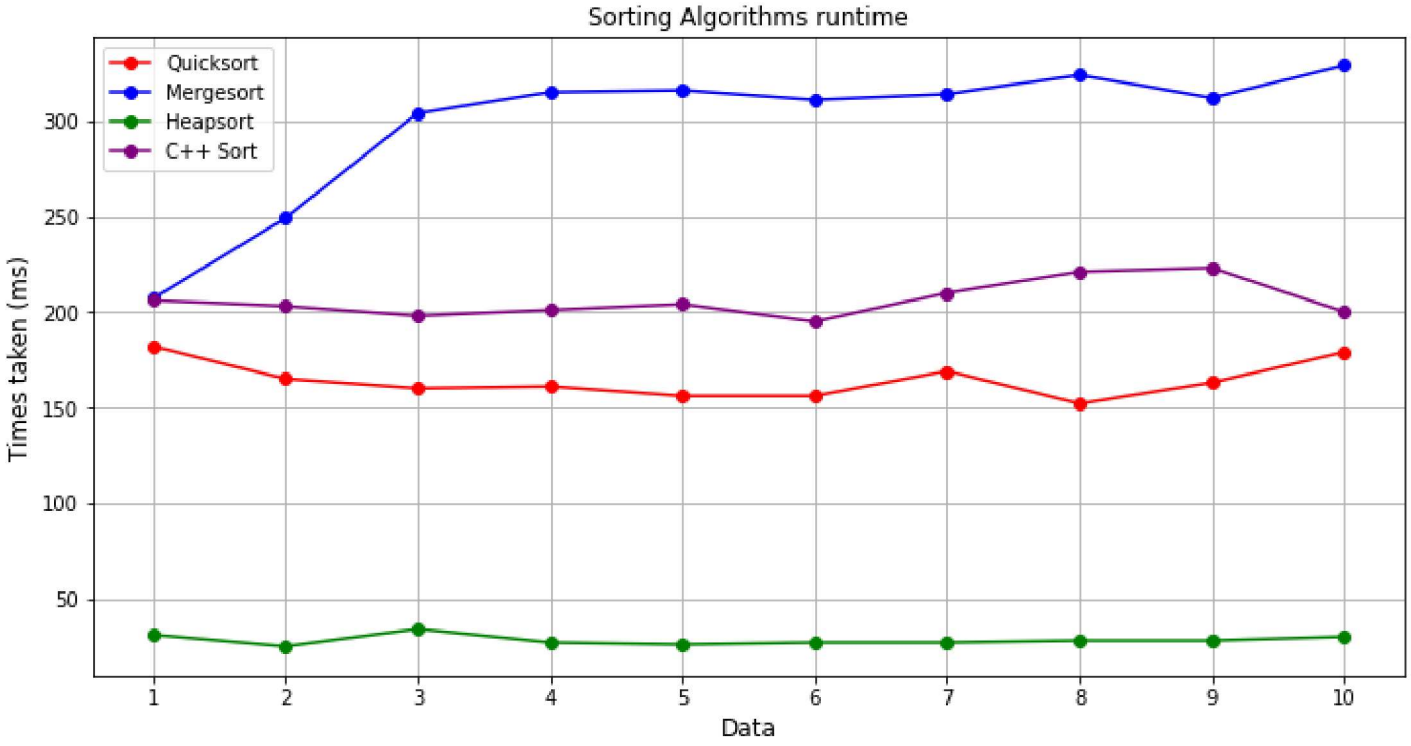
Nhìn chung, các thuật toán đều xử lí rất nhanh chóng với thời gian trong vòng dưới 1s. Tuy nhiên, ta vẫn có thể nhận ra một số điểm khác biệt giữa các thuật toán như sau:

- Đúng như tên gọi của mình, Quicksort là thuật toán nhanh nhất trong bảng so sánh. Sau khi thử nhiều lần với các bộ dataset khác nhau, Quicksort vẫn luôn nhanh nhất trong tất cả 4 thuật toán. Tuy nhiên, trong trường hợp pivot xấu (khá hiếm nếu data ngẫu nhiên), Quicksort hoàn toàn có thể chạy lâu hơn (vì phải gọi đệ quy liên tục)



- Nhờ việc tối ưu tốt bằng cách kết hợp những điểm mạnh của của các thuật toán khác, C++ Sort luôn cho ra kết quả khá ổn, trung bình rằng C++ Sort cần khoảng 280ms để hoàn thành việc sắp xếp. Ngoài ra, C++ Sort rất ổn định khi thử nhiều bộ dataset khác nhau
- Mergesort cũng tương tự như thế, đều cho ra kết quả ổn định với các bộ dữ liệu. Tuy nhiên, nếu ta giảm số lượng input xuống tầm 1000 thay vì 1 triệu thì mergesort lại thực thi rất lâu.
- Trái ngược với Mergesort, Heapsort tỏ ra hiệu quả hơn khi chạy với số lượng đầu vào thấp với thời gian thực thi dưới 50ms

Kết quả sau khi thử nghiệm với 1000 input trong mỗi bộ data:



IV. Tổng kết

Nhìn chung, cả 4 thuật toán được khảo sát đều là những thuật toán sắp xếp rất nhanh và hiệu quả. Tuy nhiên, mỗi cách sắp xếp đều có những nhược điểm của riêng mình. Chính vì vậy, ta cần phải chọn ra thuật toán phù hợp với nhu cầu của mình để tối ưu hóa thời gian chạy và bộ nhớ cần sử dụng:

- *Quicksort* : các đầu vào ngẫu nhiên, không cần nhiều bộ nhớ và chạy những chương trình cần tốc độ.
- *Mergesort*: các input có quy luật và với số lượng lớn, khi cần độ ổn định, hoặc khi sắp xếp các mảng thường rơi vào trường hợp xấu.
- *Heapsort*: các mảng đã được sắp xếp một phần, input bị giới hạn giá trị hoặc sắp xếp với số lượng ít.
- *C++ Sort*: Được sử dụng rộng rãi trong các thư viện lập trình. Vì vậy, đây có thể coi là thuật toán đáng tin cậy dành cho việc sắp xếp.