1 Code Tuning

Trong bài tập 2, nhóm đã viết chương trình giải mô phỏng thuật toán Merge Sort trên hai ngôn ngữ Java và Python, cũng như thực hiện các phương pháp kiểm thử tương ứng đối đối với hai mã nguồn này. Trong bài tập lần này, nhóm tiếp tục sử dụng các mã nguồn trên để thực hiện tối ưu hóa hiệu năng của chương trình.

1.1 Một số thay đổi trước khi bắt đầu bài tập

Chương trình mô phỏng thuật toán Merge Sort bao gồm hai module là hàm merge() và hàm sort(). Cả hai hàm này đều được đặt quyền truy cập public, nhằm cho phép các chương trình kiểm thử truy cập từ bên ngoài.

Tuy nhiên trong quá trình sử dụng, hàm merge() chỉ cần được gọi như là một bước con của thuật toán Merge Sort. Do đó trong bài tập lần này, quyền truy cập của merge() được đặt lại làm private. Điều này giúp giảm số lượng trường hợp cần phải xử lý, và làm cho ví dụ trở nên thực tế hơn.

1.2 Thực hiện các kỹ thuật hiệu chỉnh mã nguồn

1.2.1 Hiệu chỉnh các biểu thức logic

Trong quá trình kiểm thử, hàm merge() và hàm sort() đều có các điều kiểm tra mảng được truyền vào. Trong trường hợp mảng truyền vào là null, hoặc có độ dài không quá 1, chương trình sẽ lập tức thoát ra, không phải xử lý gì cả.

```
if (arr == null || arr.length <= 1)
    return arr;
```

Tuy nhiên đoạn code kiểm tra này chỉ nên được gọi một lần, khi mảng được truyền vào lần đầu tiên. Trong các lần gọi tiếp theo của vòng lặp đệ quy, mảng được truyền vào được tính toán trực tiếp từ bên trong code, và không thể xảy ra trường hợp bằng null.

Ảnh hưởng về hiệu năng lên chương trình càng lớn hơn trong chương trình Python: do Python không định kiểu chặt lên biến, các thủ tục phải thực hiện kiểm tra dữ liệu đầu vào có đúng kiểu hỗ trợ hay không, trước khi tiếp tục xử lý. Việc kiểm tra này mất thời gian tuyến tính với độ dài của mảng, và làm giảm hiệu suất chương trình một cách đáng kể:

```
def __validate_(arr):
    if (type(arr) is not list):
        return False
    for x in arr:
        if (type(x) is not int):
            return False
    return True
```

Phương án cải thiện: Tạo thêm một thủ tục sortRecursive(), đảm nhiệm việc tính toán một lần đệ quy của thuật toán sắp xếp. Việc kiểm tra null và kiểm tra kiểu dữ liệu chỉ được thực hiện một lần duy nhất tại hàm sort(). Sau khi kiểm tra xong, hàm sort() gọi sortRecursive(), thực hiện nội dung chính của thuật toán. Điều này vừa giúp cải thiện hiệu năng mà không làm thay đổi interface của chương trình.

```
public static void sort(int[] arr) {
    if (arr == null)
        return;
    cache = new int[arr.length];
    sortRecursive(...);
}
```

1.2.2 Hiệu chỉnh vòng lặp

Trong hàm merge(), việc ghép hai nửa đã sắp xếp của mảng được thực hiện bằng cách duyệt qua tất cả các vị trí của mảng kết quả, và gán giá trị tại vị trí đó bằng giá trị nhỏ hơn trong hai giá trị đứng đầu mỗi nửa. Nếu như một trong hai nữa đã hết giá trị thì mặc định là gán bằng giá trị của nửa còn lại.

```
for(int i = 0; i < result.length; i ++) {
    if (idLeft == left.length)
        result[i] = right[idRight ++];
    else if (idRight == right.length)
        result[i] = left[idLeft ++];
    else {
        if (left[idLeft] < right[idRight])
            result[i] = left[idLeft ++];
        else
            result[i] = right[idRight ++];
    }
}
```

Trong trường hợp nửa bên phải hết trước, điều kiện kiểm tra vẫn phải được thực hiện đối với nửa bên trái trước (và chắc chắn trả về đúng) khi thực hiện với nửa bên phải. Điều này làm ảnh hưởng tới hiệu năng của chương trình.

Phương án cải thiện: Thay một vòng lặp for bằng 3 vòng lặp while: một vòng lặp chạy khi cả hai nửa còn phần tử, và 2 vòng lặp chạy để đưa các phần tử còn lại vào cuối kết quả:

```
while(idLeft < middle && idRight < to) {
    if (cache[idLeft] < cache[idRight])
        arr[iterPos ++] = cache[idLeft ++];
```

```
        else
            arr[iterPos ++] = cache[idRight ++];
    }
    while(idLeft < middle)
        arr[iterPos ++] = cache[idLeft ++];
    while(idRight < to)
        arr[iterPos ++] = cache[idRight ++];
```

Điều này giúp loại bỏ công việc thừa thãi trong vòng lặp, làm tăng tốc độ chương trình.

1.2.3 Hiệu chỉnh chuyển đổi dữ liệu

Do chương trình chỉ thao tác so sánh giữa các giá trị thuộc một kiểu dữ liệu duy nhất (số nguyên) nên không phát sinh việc chuyển đổi dữ liệu.

1.2.4 Hiệu chỉnh các biểu thức

Thuật toán Merge Sort cần thực hiện tính toán vị trí chính giữa của các đoạn để chia đoạn cần sắp xếp làm hai.

```
int middle = arr.length / 2;
```

Phương án cải thiện: Phép toán chia một số nguyên cho hai được thực hiện rất nhiều lần, và có thể được thay thế bằng phép dịch phải sang 1 để tăng cường hiệu năng.

```
int middle = from + (length >> 1);
```

1.2.5 Hiệu chỉnh hàm/thủ tục

Trong chương trình ban đầu, các thao tác tính toán trả về các object thuộc kiểu mảng số nguyên: việc chia mảng làm đôi được thực hiện bằng cách copy giá trị của mỗi nửa để tạo thành hai object mới; việc ghép hai mảng đã sắp xếp cũng trả về một object mảng, vân vân... Cách cài đặt này rất tiện lợi cho việc đọc hiểu, vì chỉ sử dụng các hàm được cung cấp sẵn bởi ngôn ngữ, và phải thực hiện rất ít thao tác toán.

```
int[] left = sort(
    Arrays.copyOfRange(arr, 0, middle)
);
int[] right = sort(
    Arrays.copyOfRange(arr, middle, arr.length)
);
int[] result = merge(left, right);
```

Tuy nhiên, việc tạo thêm object mới liên tục khiến cho mức tiêu thụ bộ nhớ của thuật toán trở thành $O(nlog(n))$, cao hơn so với mức chuẩn là $O(n)$. Điều này vừa khiến chương trình trở nên lãng phí tài nguyên hơn, lại vừa làm giảm hiệu năng do phải liên tục kêu gọi cấp phát bộ nhớ cho các object mới.

Phương án cải thiện: chỉnh sửa hàm sortRecursive() và merge() thành thao tác theo vị trí trên mảng được truyền vào, thay vì thao tác trên object. Ngoài ra trong hàm sort(), ta cũng khởi tạo một mảng số nguyên cache[] có cùng kích cỡ với mảng truyền vào để lưu trữ giá trị trong khi chạy thuật toán. Nhờ đó, ta loại bỏ đi được việc cần phải yêu cầu bộ nhớ động.

```
public static void sort(int[] arr) {
    if (arr == null)
        return;
    cache = new int[arr.length];
    sortRecursive(arr, 0, arr.length);
}
```

1.3 Kiểm tra cải thiện hiệu năng

1.3.1 Môi trường kiểm thử

- Hệ điều hành: Antergos Linux
- Phiên bản Kernel: 5.0.5-arch1-1-ARCH
- Kiến trúc: 64-bit
- Vi xử lý: Intel® Core™ i3-3217U CPU, 4 nhân, 1.80GHz
- Bộ nhớ: 7.7 GiB of RAM
- Phiên bản Java: OpenJDK Runtime Environment 11.0.3
- Phieen bản Python: Python 3.7.3

Các chương trình đọc dữ liệu từ file test.txt, chứa một dãy các số nguyên được sinh ngẫu nhiên theo phân phối chuẩn. Chương trình Java sẽ thực hiện sắp xếp dãy số 10^7 phần tử, trong khi chương trình Python sẽ thực hiện trên dãy số 10^6 phần tử. Điều này là do Python là ngôn ngữ thông dịch, và do đó có tốc độ chậm hơn so với ngôn ngữ biên dịch là Java.

Thời gian hoạt động của thuật toán được tính duy nhất trên hàm sort() - bỏ qua các giai đoạn nhập và xuất dữ liệu. Đối với Java, việc tính toán được thực hiện như sau:

```
long startTime = Instant.now().toEpochMilli();
sort(arr);
long endTime = Instant.now().toEpochMilli();
```

```
long runTime = endTime - startTime;
System.out.println(runTime);
```

Đối với Python:

```
start_time = datetime.datetime.now()
sort(array)
end_time = datetime.datetime.now()
print(end_time - start_time)
```

Nhóm cũng sử dụng timeout, một chương trình Perl nhỏ cho phép đo đạc lại thời gian hoạt động và lượng tài nguyên sử dụng của chương trình command line trên Linux.

1.3.2 Số liệu đo đạc

Ngôn ngữ	Thời gian chạy trước tối ưu	Thời gian chạy sau tối ưu	Mức độ cải tiến
Java	3444 ms	2907 ms	39.11%
Python	2340 ms	1442 ms	38.38%

Bảng 1: Số liệu đo đạc thời gian chạy của chương trình

Ngôn ngữ	Bộ nhớ sử dụng trước tối ưu	Bộ nhớ sử dụng sau tối ưu	Mức độ cải tiến
Java	302227 byte	265681 byte	12.09%
Python	99893 byte	100211 byte	0.32%

Bảng 2: Số liệu đo đạc thời gian chạy của chương trình

1.3.3 Nhận xét

Phần lớn các cải tiến được đề xuất trong báo cáo đã cho thấy hiệu quả rõ rệt: Hiệu năng của chương trình được cải thiện tới gần 40%. Mức độ yêu cầu bộ nhớ Java giảm hơn 12%. Tuy nhiên, vẫn xảy ra ngoại lệ trên số liệu về bộ nhớ của Python.