ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

MÔN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

LAB 02: ĐỘ PHỨC TẠP THUẬT TOÁN

Lớp 22_7

Giáo viên hướng dẫn:	**Trần Hoàng Quân**

MỤC LỤC

PHÂN CÔNG

NHÓM 5		
Họ và tên	**MSSV**	**Mức độ hoàn thành**
Nguyễn Quang Thông	**22120353**	**100%**
Nguyễn Thanh Phong	**22120265**	**100%**
Võ Tuấn Tài	**22120319**	**100%**

Chú thích:

- **[x]: Lấy phần nguyên của số x**

BÁO CÁO

1. ƯỚC LƯỢNG ĐỘ PHỨC TẠP THUẬT TOÁN

(i). sumHalf:

(i) sumHalf

```
int sumHalf(int n) {
    int a = 0, i = n;
    while (i > 0) {
        a = a + i;
        i = i / 2;
    }
    return a;
}
```

Mã nguồn 1.1: Thuật toán sumHalf.

Trong source code, ta thêm 2 biến đếm là count_assignments và count_comparisons để đếm số lần gán và số lần so sánh của thuật toán qua mỗi trường hợp:

```
// ước lượng độ phức tạp thuật toán
// ý (i)

int sumHalf(int n, int& count_comparisons,int& count_assignments)
{
    count_comparisons = 0;
    count_assignments = 0;

    int a = 0; count_assignments++;
    int i = n; count_assignments++;
    while (++count_comparisons && i > 0)
    {
        a = a + i;
        count_assignments++;
        i = i / 2;
        count_assignments++;
    }
    return a;
}
```

Mã nguồn 1.2:Thuật toán sumHalf đã chèn thêm các biến đếm số phép gán và số phép so sánh vào các vị trí thích hợp.

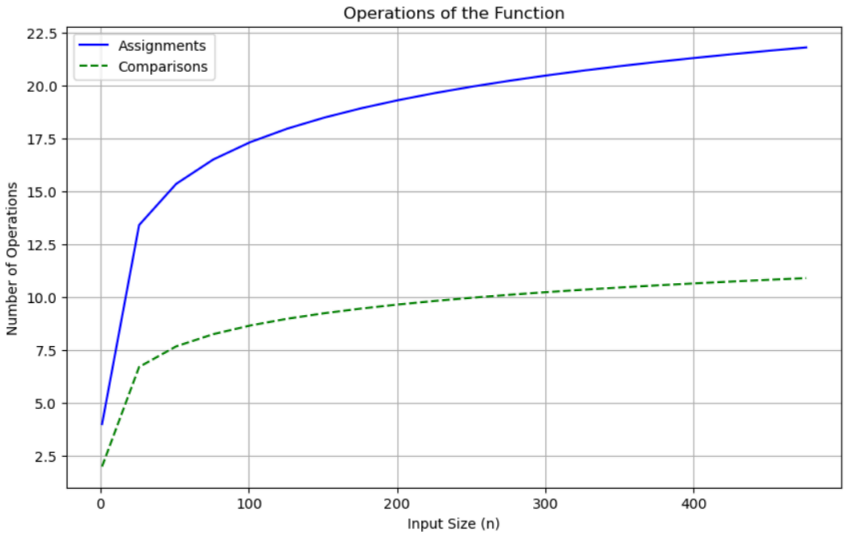
Ban đầu, ta khởi tạo 2 biến đếm = 0.

+ int a = 0 ⇒ ta thực hiện gán a = 0 nên tăng biến đếm count_assignments lên 1
+ int i = n ⇒ ta thực hiện gán i = n nên tăng biến đếm count_assignments tiếp lên 1
+ Bên trong câu điều kiện của vòng lặp, ta thêm phép ++count_comparisons để đếm số lần so sánh qua mỗi vòng lặp và so sánh điều kiện của vòng lặp (i>0)
+ Qua mỗi lần lặp, bên trong vòng lặp sẽ thực hiện 2 phép gán là a = a+i và i = i/2 nên ta thêm 2 câu lệnh count_assignments++ vào sau 2 phép gán này để đếm số phép gán.

Trực quan hóa: Sử dụng thực nghiệm, sau khi thêm các biến đếm count_comparisons và count_assignments vào các vị trí thích hợp trong thuật toán sumHalf, ta có thể trực quan hóa kết quả thông qua các bảng biểu và đồ thị như bên dưới:

N	Assignments	Comparisons
0	2	1
25	12	6
50	14	7
75	16	7
100	16	8
125	16	8
150	18	8
175	18	8
200	18	9
225	18	9
250	18	9
275	20	9
300	20	9
325	20	9
350	20	9
375	20	10
400	20	10
425	20	10
450	20	10
475	20	10
500	20	10

Bảng biểu 1.1: Số liệu trực quan hóa qua các số liệu.



Đồ thị 1.1: Sơ đồ trực quan hóa số liệu.

Nhận xét:

Xét hàm sumHalf:

1. **SỐ PHÉP SO SÁNH:**

   Gọi **S(n)** là tổng số phép so sánh

- Xét vòng lặp while:

+ Biến i = n → 0 với step: i = i/2 ⇔ biến i = $n/2^0 \rightarrow n/2^k$ với k là số mũ để vòng lặp không thỏa điều kiện
⇔ $[n/2^k] \leq 0 \Leftrightarrow 2^k > n \Leftrightarrow k > [\log_2(n)] \Rightarrow k = [\log_2(n)] + 1$

+ Gọi i' là số mũ của 2, $i' \in [0,k] \Rightarrow$ vòng lặp sẽ chạy từ i' = 0 đến i' = k - 1 = $[\log_2(n)]$ và dừng khi i' = k = $[\log_2(n)] + 1$

⇒ Vòng lặp while sẽ được thực hiện $[\log_2(n)] + 1$ lần

⇒ số phép so sánh được thực hiện $[\log_2(n)] + 2$ lần ⇒ $S(n) = [\log_2(n)] + 2$

2. SỐ PHÉP GÁN:

Gọi T(n) là tổng số phép gán

- Bên ngoài vòng while sẽ có 2 phép gán cố định (1)
- Xét vòng lặp while, theo như tính toán ở số phép so sánh, vòng lặp while sẽ được thực hiện $[\log_2(n)] + 1$ lần. (2)
- Trong vòng while, chúng ta có 2 phép gán phụ thuộc vào vòng lặp while (3)

⇒ từ (1), (2), (3) , ta có $T(n) = 2 + 2.([\log_2(n)] + 1) = 2.[\log_2(n)] + 4$ số phép gán

Ước lượng lý thuyết (big-O)

+ Ta có $S(n) = [\log_2(n)] + 2 \leq 3[\log_2(n)], \forall n \geq 1 \Rightarrow \exists (C,k) = (3,1)$
⇒ S(n) có $O(\log_2(n))$
+ Ta có $T(n) = 2.[\log_2(n)] + 4 \leq 6[\log_2(n)] \quad \forall n \geq 1 \Rightarrow \exists (C,k) = (3,1)$
⇒ T(n) có $O(\log_2(n))$

⇒ Hàm tăng tuyến tính theo $\log_2(n)$ với số phép gán là $2.[\log_2(n)] + 4$ và số phép so sánh là $[\log_2(n)] + 2$ ⇒ hàm có $O(\log_2(n))$.

(ii) recursiveSquareSum:

```
26 int recursiveSquareSum(int n){
27     if(n < 1) return 0;
28     return n * n + recursiveSquareSum(n - 1);
29 }
```

Mã nguồn 1.3: Thuật toán recursiveSquareSum.

- Theo lý thuyết ta biết thuật toán này có độ phức tạp tuyến tính là O(n), có đúng n phép gán và 2n+1 phép so sánh. Để kiểm chứng, ta thêm các biến đếm số phép gán, số phép so sánh count_assigment, count_comparison vào vị trí thích hợp trong mã nguồn 1.3.

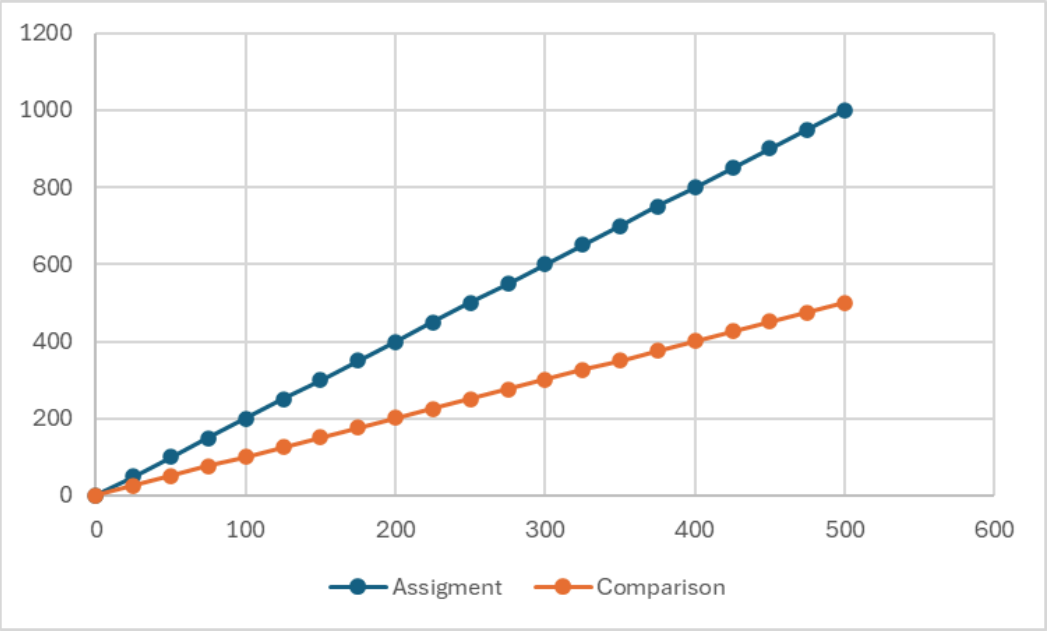
```
20 int recursiveSquareSum(int n, int& count_comparison, int& count_assigment){
21     ++count_comparison;
22     if(n < 1) return 0;
23     ++count_assigment;
24     ++count_assigment;
25     return n * n + recursiveSquareSum(n - 1, count_comparison, count_assigment);
26 }
```

Mã nguồn 1.4: Thuật toán recursiveSquareSum, đã chèn thêm các biến đếm số phép gán và số phép so sánh vào các vị trí thích hợp.

- **Trực quan hóa** Sử dụng thực nghiệm, sau khi thêm các biến đếm *count_comparisons* và *count_assignments* vào các vị trí thích hợp trong thuật toán sumHalf, ta có thể trực quan hóa kết quả thông qua các bảng biểu và đồ thị như bên dưới:

n	Assigment	Comparison
0	0	1
25	50	26
50	100	51
75	150	76
100	200	101
125	250	126
150	300	151
175	350	176
200	400	201
225	450	226
250	500	251
275	550	276
300	600	301
325	650	326
350	700	351
375	750	376
400	800	401
425	850	426
450	900	451
475	950	476
500	1000	501

Bảng biểu 1.2: Số liệu trực quan hóa qua các số liệu.



Sơ đồ 1.2: Sơ đồ trực quan hóa số liệu.

2. THIẾT KẾ GIẢI THUẬT TOÁN

(i). Ước chung lớn nhất:

Thuật toán 1.

Dữ liệu đầu vào của thuật toán này là hai số nguyên u và v ngẫu nhiên, và thuật toán sẽ trả về ước chung lớn nhất của chúng. Thuật toán Euclid hoạt động tốt với các bộ dữ liệu có kích thước lớn và có độ phức tạp thời gian thấp hơn.

```
int gcdEuclid(int u, int v) {
    while (v != 0) {
        int temp = u % v;
        u = v;
        v = temp;
    }
    return u;
}
```

Mã nguồn 2.1: Thuật toán tìm ước chung lớn nhất bằng giải thuật Euclid của 2 số u và v.

Theo lý thuyết: ta đã biết đoạn thuật toán này có độ phức tạp O(log(min(u,v))). Để kiểm chứng, ta dùng đoạn mã nguồn 2.1, với các biến đếm số phép gán count_assignments và đếm số phép so sánh count_comparisons được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

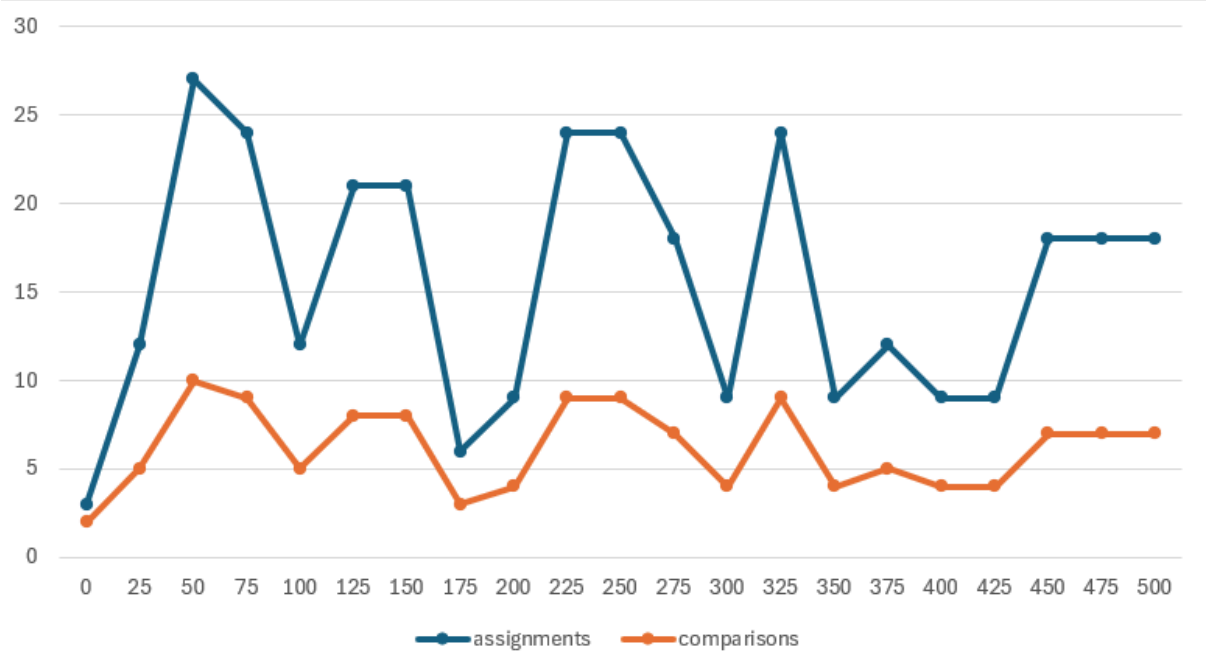
```
int gcdEuclid(int u, int v, int& count_assignments, int& count_comparisons) {
    count_assignments = 0;
    count_comparisons = 0;

    while ((++count_comparisons) && v != 0) {
        int temp = u % v;
        u = v;
        v = temp;
        count_assignments += 3;
    }
    return u;
}
```

Mã nguồn 2.2: Thuật toán tìm ước chung lớn nhất bằng giải thuật Euclid của 2 số u và v, đã chèn các biến đếm số phép gán và phép so sánh.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán gcdEuclid(u,v), ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

u	v	assignments	comparisons
0	420	3	2
25	295	12	5
50	129	27	10
75	179	24	9
100	399	12	5
125	316	21	8
150	87	21	8
175	2	6	3
200	75	9	4
225	412	24	9
250	414	24	9
275	114	18	7
300	303	9	4
325	456	24	9
350	54	9	4
375	108	12	5
400	336	9	4
425	125	9	4
450	494	18	7
475	61	18	7
500	303	18	7

Bảng biểu 2.1: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.1: Sơ đồ trực quan hóa số liệu.

Nhận xét:

- Trường hợp tốt nhất của thuật toán Euclid xảy ra khi v có giá trị bằng 0 từ đầu. Trong trường hợp này, vòng lặp sẽ không thực hiện bất kỳ lần nào và thuật toán sẽ trả về giá trị của u ngay từ lúc ban đầu. Điều này xảy ra khi v là 0 hoặc v là ước số chung lớn nhất của u và v. Do đó, độ phức tạp của trường hợp tốt nhất là O(1).
- Trong trường hợp tệ nhất của thuật toán Euclid, u và v là các số nguyên dương rất lớn và không cùng với nhau. Số lần lặp sẽ tăng lên tương ứng với kích thước của các số này. Trong trường hợp tồi nhất, số lần lặp sẽ tương tự với chiều dài của số v, vì u sẽ luôn bằng v sau mỗi bước lặp. Tuy nhiên, điều này chỉ xảy ra trong trường hợp cực kỳ đặc biệt, khi v là một số nguyên dương lớn nhất có thể và u là số nguyên dương nhỏ hơn nhưng vẫn lớn đủ để chia hết cho v. Trong hầu hết các trường hợp thực tế, thuật toán Euclid sẽ hoàn thành nhanh chóng với một số lần lặp nhỏ hơn. Do đó, độ phức tạp của trường hợp tệ nhất là O(logmin(u,v)).

Thuật toán 2.

Dữ liệu đầu vào của thuật toán này là hai số nguyên u và v ngẫu nhiên, và thuật toán sẽ trả về ước chung lớn nhất của hai số này. Tuy nhiên, đặc điểm của thuật toán này là không hiệu quả với các bộ dữ liệu có kích thước lớn hoặc với các bộ dữ liệu mà min(u,v) là một số lớn.

```
int gcd(int u, int v) {
    int gcd = 1;
    int limit = min(u, v);
    for (int i = 2; i <= limit; i++) {
        if (u % i == 0 && v % i == 0) {
            gcd = i;;
        }
    }
    return gcd;
}
```

Mã nguồn 2.3: Thuật toán tìm ước chung lớn nhất của 2 số u và v.

Theo lý thuyết: ta đã biết đoạn thuật toán này có độ phức tạp O(min(u,v)). Để kiểm chứng, ta dùng đoạn mã nguồn 2.3, với các biến đếm số phép gán count_assignments và đếm số phép so sánh count_comparisons được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

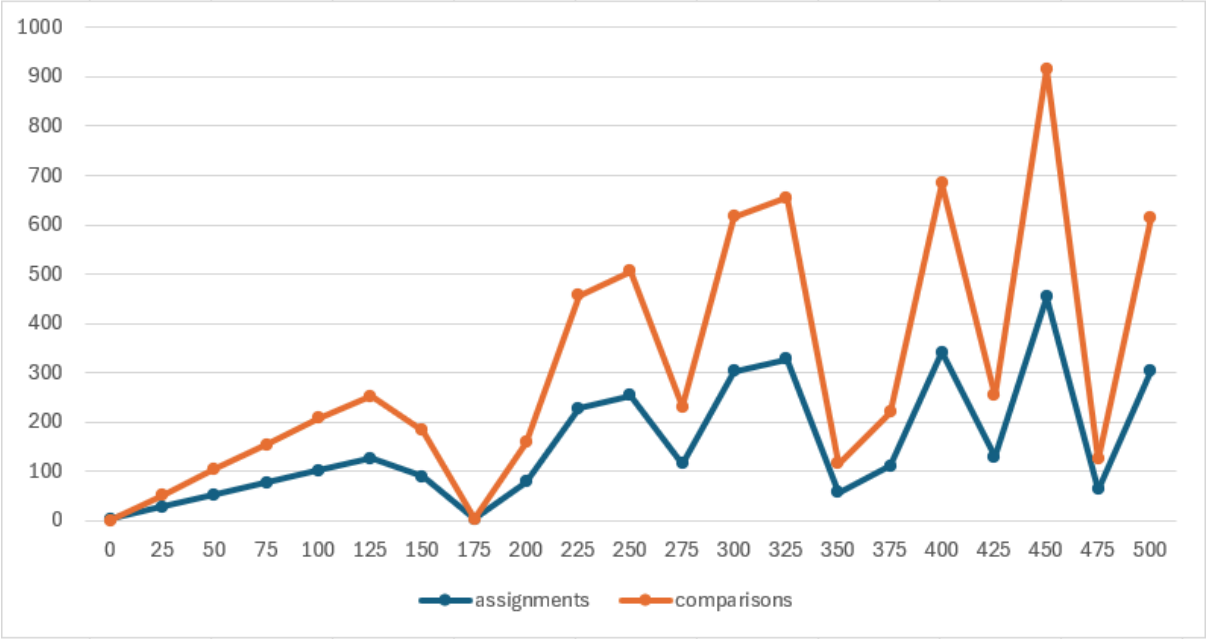
```cpp
int gcd(int u, int v, int& assignments, int& comparisons) {
    assignments = 0;
    comparisons = 0;
    int gcd = 1; assignments++;
    int limit = min(u, v); assignments++;
    int i = 2; assignments++;
    for (; ++comparisons && i <= limit; i++) {
        assignments++;
        if ((++comparisons && u % i == 0) && (++comparisons && v % i == 0)) {
            gcd = i;
            assignments++;
        }
    }
    return gcd;
}
```

Mã nguồn 2.4: Thuật toán tìm ước chung lớn nhất của 2 số u và v, đã chèn các biến đếm số phép gán và phép so sánh.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán gcd(u,v), ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

u	v	assignments	comparisons
0	420	3	1
25	295	28	51
50	129	52	104
75	179	77	154
100	399	102	207
125	316	127	252
150	87	90	183
175	2	4	3
200	75	79	158
225	412	227	457
250	414	253	506
275	114	116	231
300	303	303	616
325	456	327	654
350	54	57	115
375	108	111	220
400	336	342	684
425	125	129	253
450	494	453	916
475	61	63	124
500	303	305	615

Bảng biểu 2.2: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.2: Sơ đồ trực quan hóa số liệu.

Nhận xét:

- Trường hợp tốt nhất xảy ra khi u và v không có ước số chung nào, tức là khi u và v là các số nguyên tố cùng nhau. Khi đó, vòng lặp for sẽ chỉ thực hiện một số lần lặp tương ứng với giá trị của biến limit (vì không có ước số chung nào), và không có bước nào trong vòng lặp thỏa mãn điều kiện kiểm tra ước số chung.Do đó, độ phức tạp trong trường hợp tốt nhất là O(1).
- Trong trường hợp xấu nhất, u và v là hai số nguyên lớn và có nhiều ước số chung. Trong trường hợp này, vòng lặp for sẽ thực hiện limit−1 lần (với limit là số nhỏ nhất giữa u và v), và mỗi lần lặp sẽ kiểm tra xem có phải là ước số chung không. Do đó, độ phức tạp trong trường hợp xấu nhất là O(min(u,v)).

<u>Nhận xét chung:</u>

Ta có thể thấy Thuật toán 1 tối ưu hơn Thuật toán 2 do độ phức tạp của Thuật toán 1 là O(log(min(u,v))) nhỏ hơn độ phức tạp của Thuật toán 2 cũng như số lần gán và so sánh của Thuật toán 1 nhỏ hơn Thuật toán 2 nếu xét cùng u và v.

(ii). Dãy con liên tiếp không giảm dài nhất:

Thuật toán 1.

Đặc điểm của dữ liệu đầu vào của thuật toán này bao gồm:

- Số nguyên n biểu diễn kích thước của mảng, tức là số lượng phần tử trong mảng.
- Một mảng số nguyên a có n phần tử, mỗi phần tử đại diện cho một số nguyên trong dãy trong đó $-10^5 \leq a_i \leq 10^5$

```cpp
int longestNonDecreasingSubsequenceLength1(int* a, int n) {
    int*dp= new int[n];
    for (int i = 0; i < n; ++i) {
        dp[i] = 1;
    }
    for (int i = 1; i < n; ++i) {
        if (a[i] >= a[i - 1]) {
            dp[i] = dp[i - 1] + 1;
        }
    }
    int maxLength = 0;
    for (int i = 0; i < n; ++i) {
        maxLength = max(maxLength, dp[i]);
    }
    return maxLength;
}
```

Mã nguồn 2.5: Thuật toán tìm độ dài của dãy con liên tiếp không giảm dài nhất.

Theo lý thuyết: ta đã biết đoạn thuật toán này có độ phức tạp O(n), trong đó có dùng 5n + 4 phép gán và 2n + 1 phép so sánh. Để kiểm chứng, ta dùng đoạn mã nguồn 2, với các biến đếm số phép gán count_assignments và đếm số phép so sánh count_comparisons được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

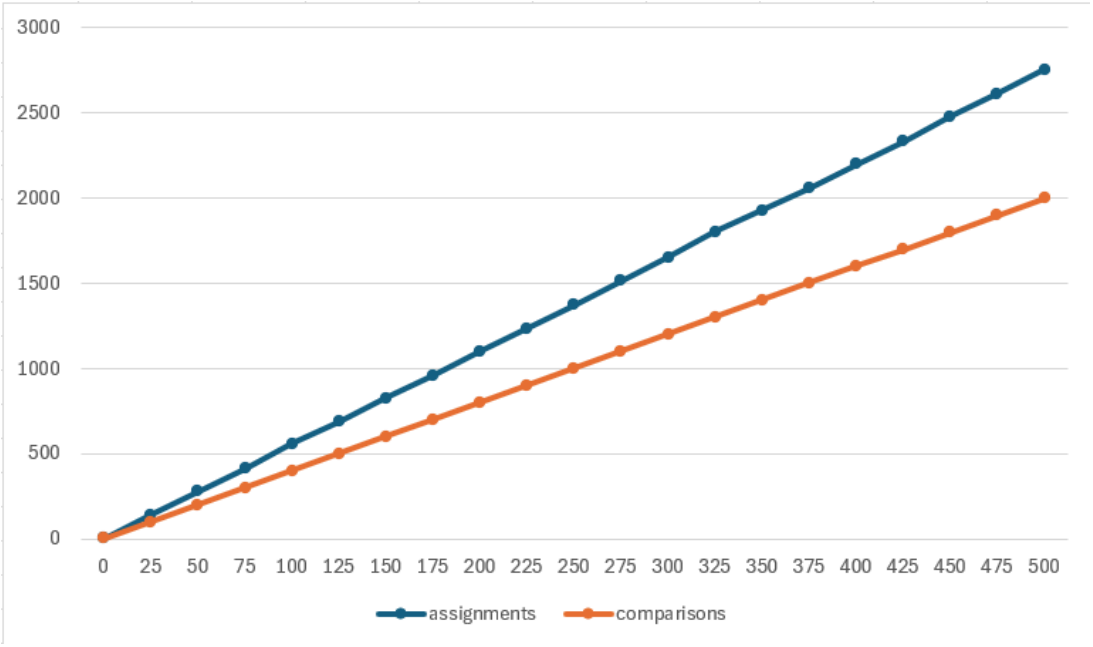
```cpp
int longestNonDecreasingSubsequenceLength1(int a[], int n, int& count_assignments, int& count_comparisons) {
    count_assignments = 0;
    count_comparisons = 0;
    int* dp = new int[n]; count_assignments++;
    int i = 0; count_assignments++;
    for (;++count_comparisons && i < n; ++i) {
        count_assignments++;
        dp[i] = 1; count_assignments++;
    }
    i = 1; count_assignments++;
    for (; ++count_comparisons && i < n; ++i) {
        count_assignments++;
        if ((count_comparisons++) && a[i] >= a[i - 1]) {
            dp[i] = dp[i - 1] + 1; count_assignments++;
        }
    }
    int maxLength = 0; count_assignments++;
    i = 0; count_assignments++;
    for (; ++count_comparisons && i < n; ++i) {
        count_assignments++;
        maxLength = max(maxLength, dp[i]); count_assignments++;
    }
    return maxLength;
}
```

Mã nguồn 2.6: Thuật toán tìm độ dài của dãy con liên tiếp không giảm dài nhất, đã chèn các biến đếm số phép gán và phép so sánh.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán, ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

n	assignments	comparisons
0	5	3
25	144	101
50	280	201
75	414	301
100	559	401
125	688	501
150	828	601
175	960	701
200	1104	801
225	1236	901
250	1373	1001
275	1516	1101
300	1651	1201
325	1802	1301
350	1930	1401
375	2057	1501
400	2201	1601
425	2334	1701
450	2480	1801
475	2614	1901
500	2754	2001

Bảng biểu 2.3: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.3: Sơ đồ trực quan hóa số liệu.

Nhận xét:

- Trong trường hợp tốt nhất, mảng đầu vào a[] đã được sắp xếp theo thứ tự không giảm từ trước. Điều này có nghĩa là mỗi phần tử a[i] đều lớn hơn hoặc bằng a[i−1] cho mọi i từ 1 đến n−1. Trong trường hợp này, vòng lặp kiểm tra điều kiện $a[i] \geq a[i-1]$ sẽ không bao giờ được thực hiện, vì điều kiện này luôn đúng. Do đó, mảng dp[] sẽ không thay đổi sau vòng lặp thứ hai, và giá trị trả về sẽ là chiều dài của mảng a[], tức là n. Độ phức tạp trong trường hợp tốt nhất: O(n).
- Trong trường hợp xấu nhất, mảng đầu vào a[] được sắp xếp theo thứ tự không tăng, nghĩa là mỗi phần tử a[i] đều nhỏ hơn a[i−1] cho mọi i từ 1 đến n−1. Trong trường hợp này, mỗi lần kiểm tra điều kiện $a[i] \geq a[i-1]$ đều sai, dẫn đến việc gán dp[i]=dp[i−1]+1 không bao giờ được thực hiện. Do đó, tất cả các phần tử trong mảng dp[] đều giữ giá trị là 1, và giá trị trả về sẽ là 1. Độ phức tạp trong trường hợp xấu nhất: O(n).

Thuật toán 2.

Đặc điểm của dữ liệu đầu vào của thuật toán này bao gồm:

- Một mảng số nguyên a có n phần tử, mỗi phần tử đại diện cho một số nguyên trong dãy.
- Số nguyên n biểu diễn kích thước của mảng, tức là số lượng phần tử trong mảng.

```
int longestNonDecreasingSubsequenceLength2(int* a, int n) {
    if (n == 0) return 0;
    int maxLength = 1;
    int currentLength = 1;
    for (int i = 1; i < n; ++i) {
        if (a[i] >= a[i - 1]) {
            currentLength++;
        }
        else {
            maxLength = max(maxLength, currentLength);
            currentLength = 1;
        }
    }
    maxLength = max(maxLength, currentLength);
    return maxLength;
}
```

Mã nguồn 2.7: Thuật toán tìm độ dài của dãy con liên tiếp không giảm dài nhất.

Theo lý thuyết: ta đã biết đoạn thuật toán này có độ phức tạp O(n), trong đó có dùng 3n + 3 phép gán và 2n + 1 phép so sánh. Để kiểm chứng, ta dùng đoạn mã nguồn 2.7, với các biến đếm số phép gán count_assignments và đếm số phép so sánh count_comparisons được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

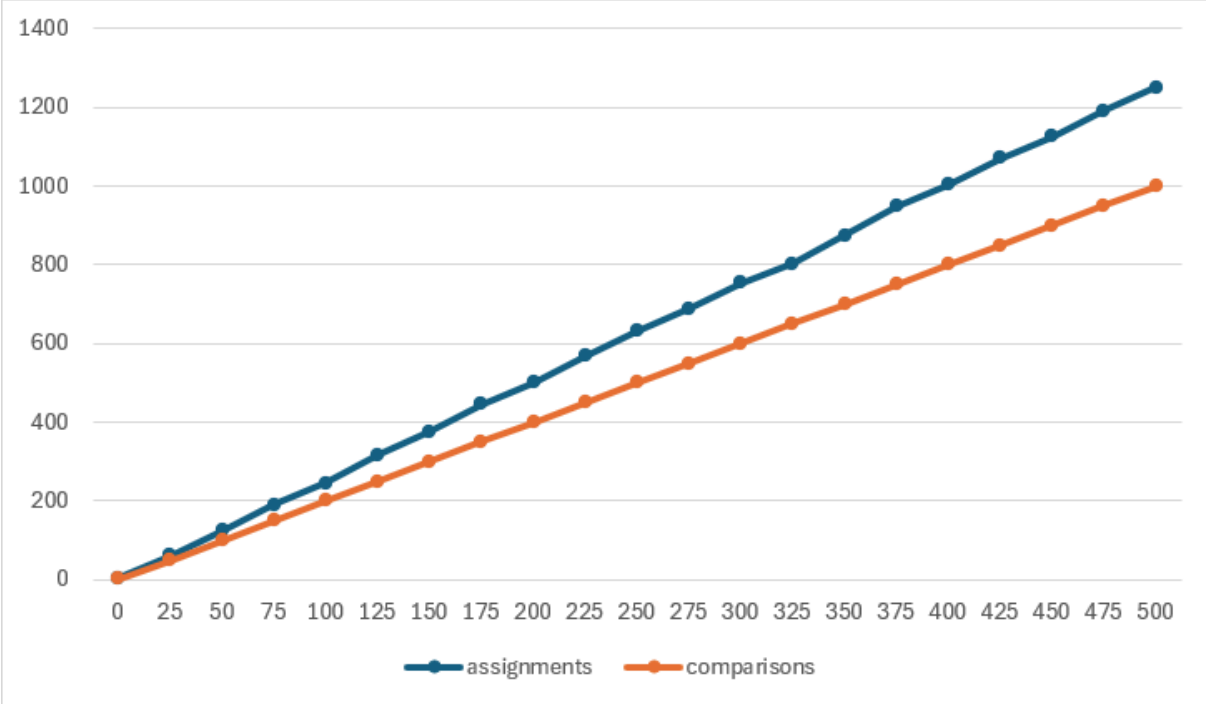
```cpp
int longestNonDecreasingSubsequenceLength2(int a[], int n, int& count_assignments, int& count_comparisons) {
    count_assignments = 0;
    count_comparisons = 0;
    if ((count_comparisons++) && n == 0) return 0;

    int maxLength = 1; count_assignments++;
    int currentLength = 1; count_assignments++;
    int i = 1; count_assignments++;
    for (;++count_comparisons && i < n; ++i) {
        count_assignments++;
        if ((count_comparisons++) && (a[i] >= a[i - 1])) {
            currentLength++;
            count_assignments++;
        }
        else {
            maxLength = max(maxLength, currentLength); count_assignments++;
            currentLength = 1; count_assignments++;
        }
    }

    maxLength = max(maxLength, currentLength); count_assignments++;
    return maxLength;
}
```

Mã nguồn 2.8: Thuật toán tìm độ dài của dãy con liên tiếp không giảm dài nhất, đã chèn các biến đếm số phép gán và phép so sánh.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán, ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

n	assignments	comparisons
0	4	2
25	61	50
50	125	100
75	191	150
100	246	200
125	317	250
150	377	300
175	445	350
200	501	400
225	569	450
250	632	500
275	689	550
300	754	600
325	803	650
350	875	700
375	948	750
400	1004	800
425	1071	850
450	1125	900
475	1191	950
500	1251	1000

Bảng biểu 2.4: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.4: Sơ đồ trực quan hóa số liệu.

Nhận xét:

- Trong trường hợp tốt nhất, mảng đầu vào a[] đã được sắp xếp theo thứ tự không giảm từ trước. Điều này có nghĩa là mỗi phần tử a[i] đều lớn hơn hoặc bằng a[i−1] cho mọi i từ 1 đến n−1. Trong trường hợp này, mỗi lần kiểm tra điều kiện a[i]≥a[i−1] đều đúng, dẫn đến việc tăng giá trị của currentLength lên. Do đó, giá trị cuối cùng của maxLength sẽ bằng chiều dài của dãy con không giảm dài nhất trong mảng a[], tức là n. Độ phức tạp trong trường hợp tốt nhất là O(1).
- Trong trường hợp xấu nhất, mảng đầu vào a[] đã được sắp xếp theo thứ tự không tăng từ trước. Điều này có nghĩa là mỗi phần tử a[i] đều nhỏ hơn a[i−1] cho mọi i từ 1 đến n−1. Trong trường hợp này, mỗi lần kiểm tra điều kiện a[i]≥a[i−1] đều sai, dẫn đến việc cập nhật maxLength nếu currentLength lớn hơn maxLength và reset currentLength về 1. Do đó, giá trị cuối cùng của maxLength sẽ là độ dài của dãy con không giảm dài nhất trong mảng a[].Độ phức tạp trong trường hợp xấu nhất là O(n).

<u>Nhận xét chung:</u>

Ta có thể thấy Thuật toán 2 tối ưu hơn Thuật toán 1 mặc dù cùng có độ phức tạp của thuật toán là O(n) nhưng số lần gán và so sánh của Thuật toán 2 nhỏ hơn Thuật toán 1 nếu xét cùng số lượng phần tử n và các phần tử trong mảng A là như nhau.

(iii). Trung vị của mảng *:

Mô tả dữ liệu đầu vào:

- Thuật toán sẽ được truyền vào ngẫu nhiên n phần tử x_i với x_i thuộc khoảng $-10^5 < x_i < 10^5$.

- Giá trị trả về của thuật toán là vector b kiểu số nguyên có n phần tử, trong đó phần tử b[i] là giá trị median của mảng a[0..i]

Thuật toán 1: Chèn giá trị x vào cuối mảng, sau mỗi lần chèn tiến hành sort ($O(n^2*logn)$)

```cpp
void heapify(vector<int>& arr, int n, int i){
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if(left < n && arr[left] > arr[largest]){
        largest = left;
    }
    if(right < n && arr[right] > arr[largest]){
        largest = right;
    }
    if(largest != i){
        int tmp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = tmp;
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr, int n){ // O(nlogn)
    // build max heap
    for(int i = n/2 - 1; i >= 0; i--){
        heapify(arr, n, i);
    }
    // sort
    for(int i = n - 1; i>= 0; i--){
        int tmp = arr[0];
        arr[0] = arr[i];
        arr[i] = tmp;
        heapify(arr, i, 0);
    }
}
```

```cpp
vector<int> findMedian_otm(vector<int> a, int n){ //O(n^2*logn)
    vector<int> b;
    int i = 0;
    while(i < n){
        int x = generateRandomNumber(-pow(10,5), pow(10,5));
        a.push_back(x);
        heapSort(a, i+1);
        if(a.size() % 2 == 0){
            b.push_back((a[a.size()/2] + a[a.size()/2 - 1])/2);
        }
        else {
            b.push_back(a[a.size()/2]);
        }
        ++i;
    }
    return b;
}
```

Mã nguồn 2.9: Thuật toán findMedian_otm sử dụng heapSort.

Ta dùng đoạn mã nguồn 2.9, với các biến đếm số phép gán asgm và đếm số phép so sánh cmp được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

```cpp
void heapify(vector<int>& arr, int n, int i, int& cmp, int& asgm){
    int largest = i; ++asgm;
    int left = 2*i + 1; ++asgm;
    int right = 2*i + 2; ++asgm;
    if(++cmp && ++cmp && left < n && arr[left] > arr[largest]){
        largest = left; ++asgm;
    }
    if(++cmp && ++cmp && right < n && arr[right] > arr[largest]){
        largest = right; ++asgm;
    }
    if(++cmp && largest != i){
        int tmp = arr[i]; ++asgm;
        arr[i] = arr[largest]; ++asgm;
        arr[largest] = tmp; ++asgm;
        heapify(arr, n, largest, cmp, asgm);
    }
}

void heapSort(vector<int>& arr, int n, int& cmp, int& asgm){ // O(nlogn)
    int i = n/2 - 1; ++asgm;
    while(++cmp && i >= 0){
        heapify(arr, n, i, cmp, asgm);
        --i; ++asgm;
    }
    int j = n - 1; ++asgm;
    while(++cmp && j >= 0){
        int tmp = arr[0]; ++asgm;
        arr[0] = arr[j]; ++asgm;
        arr[j] = tmp; ++asgm;
        heapify(arr, j, 0, cmp, asgm);
        --j; ++asgm;
    }
}
```

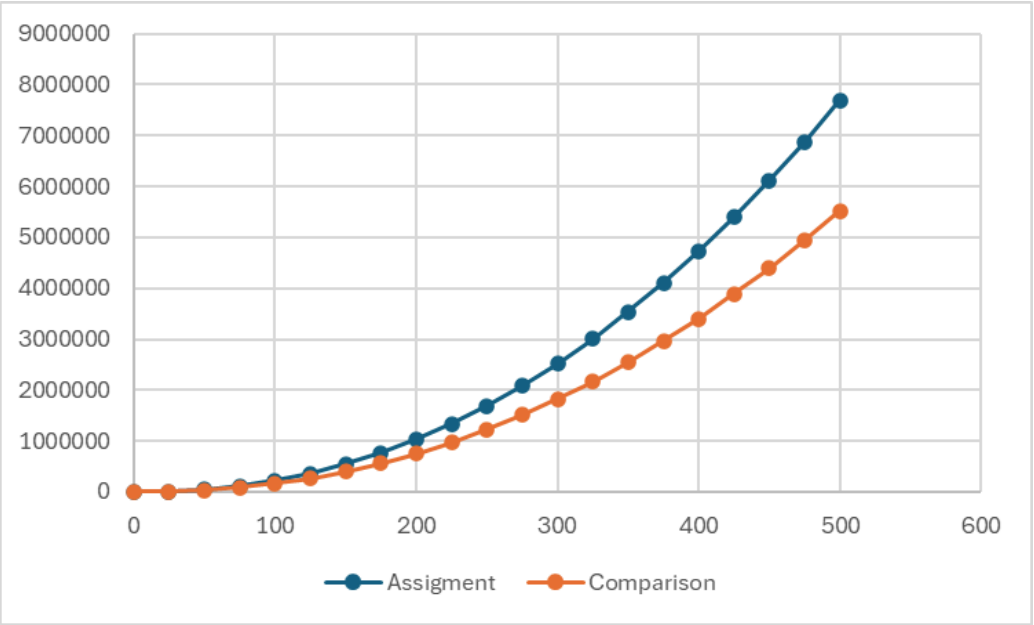
```cpp
vector<int> findMedian(vector<int> a, int n, int& cmp, int& asgm){ //O(n^2*Logn)
    vector<int> b;
    int i = 0; ++asgm;
    while(++cmp && i < n){
        int x = generateRandomNumber(-pow(10,5), pow(10,5)); ++asgm;
        a.push_back(x);
        heapSort(a, i+1, cmp, asgm);
        if(cmp++ && a.size() % 2 == 0){
            b.push_back((a[a.size()/2] + a[a.size()/2 - 1])/2);
        }
        else {
            b.push_back(a[a.size()/2]);
        }
        ++i; ++asgm;
    }
    return b;
}
```

Mã nguồn 2.10: Thuật toán findMedian_otm sử dụng heapSort, đã chèn các biến đếm số phép gán và phép so sánh.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán, ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

n	Assigment	Comparison
0	1	1
25	9398	7297
50	46155	34741
75	115491	86007
100	220711	163261
125	363779	266982
150	545152	398976
175	767725	560937
200	1032895	751771
225	1339006	973777
250	1688669	1224291
275	2080658	1506272
300	2518068	1820116
325	3001851	2167492
350	3530951	2547626
375	4108918	2960997
400	4731292	3406716
425	5400855	3886392
450	6117203	4398071
475	6880880	4943542
500	7691513	5521786

Bảng biểu 2.5: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.5: Sơ đồ trực quan hóa số liệu.

Thuật toán 2: Duyệt mảng và chèn vào vị trí thích hợp ($O(n^2)$)

- Ở thuật toán này, mỗi giá trị x_i khi thêm được thêm vào mảng a đều thỏa mãn tính tăng dần của mảng. Do đó, không cần tiến hành sort mảng a[0..i] mà chỉ cần xác định giá trị median.

```
248 vector<int> findMedian_optimize(vector<int> a, int n){
249     int i = 1;
250     vector<int> b;
251     int x;
252     if(a.empty()) {
253         x = generateRandomNumber(-pow(10,5), pow(10,5));
254         a.push_back(x);
255         }
256     while( i < n){
257         x = generateRandomNumber(-pow(10,5), pow(10,5));
258         int j = 0;
259         while(j < a.size()){
260             if(x < a[j]){
261                 a.insert( a.begin() + j, x);
262                     break;
263             }
264             ++j;
265         }
266         a.push_back(x);
267         if(a.size() % 2 == 0){
268             b.push_back((a[a.size()/2] + a[a.size()/2 - 1])/2);
269         }
270         else {
271             b.push_back(a[a.size()/2]);
272         }
273         ++i;
274     }
275     return b;
276 }
```

Mã nguồn 2.11: Thuật toán findMedian_optimize.

Ta dùng đoạn mã nguồn 2.9, với các biến đếm số phép gán asgm và đếm số phép so sánh cmp được chèn vào các vị trí thích hợp sao cho không làm thay đổi tính dung dẫn của thuật toán.

```
219 vector<int> findMedian_optimize(vector<int> a, int n, int& cmp, int& asgm){
220     int i = 1; ++asgm;
221     vector<int> b;
222     int x;
223     if(++cmp && a.empty()) {
224         x = generateRandomNumber(-pow(10,5), pow(10,5)); ++asgm;
225         a.push_back(x);
226         }
227     while(++cmp && i < n){
228         x = generateRandomNumber(-pow(10,5), pow(10,5)); ++asgm;
229         int j = 0; ++asgm;
230         while(++cmp && j < a.size()){
231             if(++cmp && x < a[j]){
232                 a.insert( a.begin() + j, x);
233                     break;
234             }
235             ++j; ++asgm;
236         }
237         a.push_back(x);
238         if(cmp++ && a.size() % 2 == 0){
239             b.push_back((a[a.size()/2] + a[a.size()/2 - 1])/2);
240         }
241         else {
242             b.push_back(a[a.size()/2]);
243         }
244         ++i; ++asgm;
245     }
246     return b;
247 }
```

Mã nguồn 2.12: Thuật toán findMedian_optimize, đã chèn các biến đếm số phép gán và phép so sánh.

- Ở thuật toán này độ phức tạp trung bình của thuật toán là $O(n^2)$.
- Số phép so sánh:

+ Luôn có 1 phép so sánh cố định trong câu lệnh if để kiểm tra vector a rỗng hay không (dòng 223) (1)
+ Ở vòng lặp while đầu tiên (dòng 227): i sẽ chạy từ 1 đến n-1, điều kiện dừng là i < n
  ⇒ vòng lặp while đầu tiên có n phép so sánh.(2)
+ Ở vòng lặp while thứ hai (vòng lặp bên trong, dòng 230), với mọi i ∈ [1;n-1], j sẽ chạy từ 0 đến a.size() - 1 (a.size() = i) , tức là duyệt tuần từ đầu đến cuối vector a, điều kiện dừng là j < a.size(), bên trong vòng lặp này câu điều kiện if để so sánh giá trị x với a[j], nếu x nhỏ hơn giá trị a[j] thì chèn x vào vị trí j và thoát khỏi vòng lặp. Trong trường hợp tệ nhất, giá trị x lớn hơn tất cả giá trị trong vector a, tức là phải duyệt từ đầu đến cuối vector a, số lần so sánh là i + 1 lần, trong đó i lần so sánh đúng ⇒ câu lệnh if cũng thực hiện i lần. Thoát khỏi vòng lặp và chèn x vào cuối vector (giá trị x là lớn nhất).
  ⇒ vòng lặp thứ hai có 2i + 1 phép so sánh (3)
+ Sau vòng lặp thứ hai, câu lệnh điều kiện if kiểm tra kích thước vector là chẵn hay lẻ được thực thi đúng n - 1 lần, bằng số lần so sánh đúng trong vòng lặp đầu tiên. (4)

Từ (1), (2), (3), (4) suy ra tổng số phép so sánh trong thuật toán:

$$1 + n + \sum_{i=1}^{n-1} (2i + 1) + n - 1 = \mathbf{n^2 + 2n - 1}$$

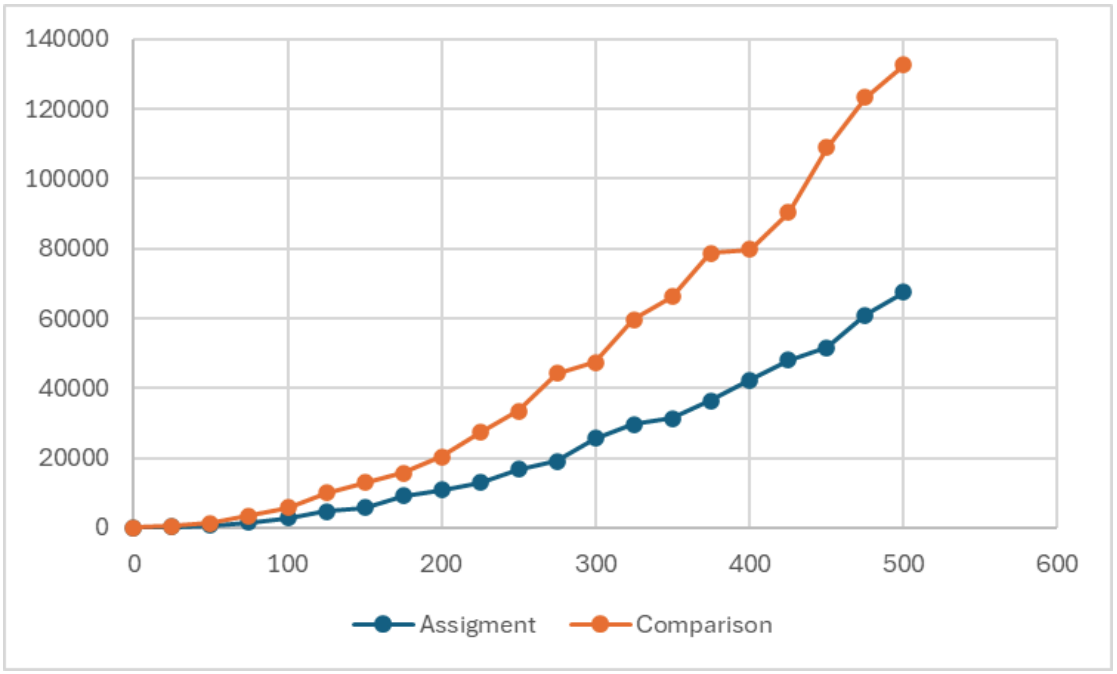
- Số phép gán:

+ Có hai phép gán cố định bên ngoài vòng lặp (gán i = 0 ở dòng 220, gán x bằng giá trị random ở dòng 224). (1)
+ Ở vòng lặp while đầu tiên (dòng 227), i sẽ chạy từ 1 đến n => có n - 1 phép gán. (2)
+ Với mỗi giá trị i ∈[1;n-1], vòng lặp while thứ hai (dòng 230), giá trị j sẽ chạy từ 0 đến a.size() (hay j : 0 → i + 1) ⇒ có i phép gán giá trị j. Ngoài ra ở các dòng 228 và 229, còn thực hiện gán giá trị x ngẫu nhiên và gán j = 0.
  ⇒ với mỗi giá trị i ∈[1;n-1], bên trong vòng lặp đầu tiên sẽ có i + 3 phép gán.
  ⇒ Tổng số phép gán: $\sum_{i=1}^{n-1} (i + 3) + 2 = \frac{n(n-1)}{2} + 3(n-1) + 2$.

Trực quan hóa: Sau khi đếm số phép gán và phép so sánh của thuật toán, ta có thể trực quan hóa kết quả qua các biểu đồ và đồ thị bên dưới:

n	Assigment	Comparison
0	2	2
25	311	540
50	811	1267
75	1464	3401
100	2774	5845
125	4701	9951
150	5871	12943
175	9258	15781
200	10910	20342
225	13007	27256
250	16853	33550
275	19197	44210
300	25693	47484
325	29665	59683
350	31408	66148
375	36386	78784
400	42283	79684
425	48044	90349
450	51579	108970
475	61000	123276
500	67391	132625

Bảng biểu 2.6: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.6: Sơ đồ trực quan hóa số liệu.

*** Nhận xét:**

- Thuật toán 1 có độ phức tạp trung bình là $O(n^2*logn)$ lớn hơn so với độ phức tạp trung bình của thuật toán 2 là $O(n^2)$
- Quan sát bảng giá trị khảo sát, ta dễ dàng thấy được số phép so sánh và số phép gán của thuật toán 2 nhỏ hơn nhiều so với số phép so sánh, số phép gán của thuật toán 1.

(**Lưu ý:** *Vì các thành phần của mảng đều được random ngẫu nhiên nên qua mỗi lần chạy, số phép gán và phép so sánh của từng thuật toán sẽ có sự thay đổi, tuy nhiên ta so sánh độ lớn giữa 2 thuật toán, có thể dự đoán được thuật toán nào tối ưu hơn*)

(iv). Số cặp nghịch thế*:

Thuật toán 1: Duyệt tuần tự ($O(n^2)$)

```
int paradoxicalNum(int arr[], int n) {
    int count = 0;
    int i = 0;
    while (i < n - 1) {
        int j = i + 1;
        while (j < n) {
            if (arr[i] > arr[j])
            {
                count++;

            }
            j = j + 1;
        }
        i = i + 1;
    }
    return count;
}
```

Mã nguồn 2.13: Thuật toán hàm paradoxicalNum sử dụng thuật toán Duyệt tuần tự.

Thuật toán 2: Merge sort (O(N.logN))

```
long long mergeSort(int arr[], int left, int right) {

    long long inv_count = 0;

    if (left < right) {
        int mid = left + (right - left) / 2;
        inv_count += mergeSort(arr, left, mid);
        inv_count += mergeSort(arr, mid + 1, right);
        inv_count += merge(arr, left, mid, right);
    }
    return inv_count;
}

long long paradoxicalNum(int arr[], int n) {
    return mergeSort(arr, 0, n - 1);
}
long long merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int* left_arr = new int[n1];
    int* right_arr = new int[n2];

    for (int i = 0; i < n1; i++)
        left_arr[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        right_arr[j] = arr[mid + 1 + j];

    long long inv_count = 0;
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k++] = left_arr[i++];
        }
        else {
            arr[k++] = right_arr[j++];
            inv_count += n1 - i;
        }
    }
```

```
    while (i < n1) {
        arr[k++] = left_arr[i++];
    }

    while (j < n2) {
        arr[k++] = right_arr[j++];
    }

    delete[] left_arr;
    delete[] right_arr;

    return inv_count;
}
```

Mã nguồn 2.14: Thuật toán hàm paradoxicalNum sử dụng thuật toán Merge sort.

Để đếm số lượng số phép gán và số phép so sánh, ta thêm các biến đếm số phép gán *count_assignments* và đếm số phép so sánh *count_comparisons* vào các vị trí thích hợp sao cho không làm thay đổi tính đúng đắn của cả 2 thuật toán:

Thuật toán 1:

```
int paradoxicalNum(int arr[], int n, int& count_assignments, int& count_comparisons) {
    count_comparisons = 0;
    count_assignments = 0;
    int count = 0;count_assignments++;
    int i = 0; count_assignments++;
    while(++count_comparisons && i < n-1) {
        int j = i + 1; count_assignments++;
        while(++count_comparisons && j < n) {
            if (++count_comparisons && arr[i] > arr[j])
            {
                count++;
                count_assignments++;
            }
            j = j + 1; count_assignments++;
        }
        i = i+1; count_assignments++;
    }
    return count;
}
```

Mã nguồn 2.15: Thuật toán hàm paradoxicalNum sử dụng thuật toán Duyệt tuần tự, sau khi thêm biến đếm số phép so sánh, số phép gán.

Thuật toán 2:

```
// Thuật toán 2
long long merge(int arr[], int left, int mid, int right, int& count_assignments, int& count_comparisons) {

    int n1 = mid - left + 1; count_assignments++;
    int n2 = right - mid; count_assignments++;

    int* left_arr = new int[n1]; count_assignments++;
    int* right_arr = new int[n2];count_assignments++;
    for (int i = 0;++count_comparisons && i < n1;i++) {
        count_assignments++;
        left_arr[i] = arr[left + i]; count_assignments++;
    }
    for (int j = 0;++count_comparisons && j < n2;j++) {
        count_assignments++;
        right_arr[j] = arr[mid + 1 + j];
        count_assignments++;
    }

    long long inv_count = 0; count_assignments++;

    int i = 0, j = 0, k = left;
    count_assignments++;
    count_assignments++;
    count_assignments++;

    while ((++count_comparisons && i < n1) && (++count_comparisons && j < n2)) {
        if (++count_comparisons && left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];
            count_assignments++;
            k++;count_assignments++;
            i++;count_assignments++;
        }
        else {
            arr[k] = right_arr[j];count_assignments++;
            k++;count_assignments++;
            j++;count_assignments++;
            inv_count += n1 - i;count_assignments++;
        }
    }
```

```
    while (++count_comparisons && i < n1) {
        arr[k] = left_arr[i];count_assignments++;
        k++;count_assignments++;
        i++;count_assignments++;
    }

    while (++count_comparisons && j < n2) {
        arr[k] = right_arr[j];count_assignments++;
        k++;count_assignments++;
        j++;count_assignments++;
    }

    delete[] left_arr;
    delete[] right_arr;

    return inv_count;
}
long long mergesort(int arr[], int left, int right, int& count_assignments,int& count_comparisons) {

    long long inv_count = 0; count_assignments++;

    if (++count_comparisons && left < right) {
        int mid = left + (right - left) / 2; count_assignments++;
        inv_count += mergesort(arr, left, mid, count_assignments, count_comparisons); count_assignments++;
        inv_count += mergesort(arr, mid + 1, right, count_assignments, count_comparisons); count_assignments++;
        inv_count += merge(arr, left, mid, right, count_assignments, count_comparisons); count_assignments++;
    }
    return inv_count;
}

long long paradoxicalnum2(int arr[], int n, int& count_assignments,int& count_comparisons) {
    return mergesort(arr, 0, n - 1, count_assignments,count_comparisons);
}
```

Mã nguồn 2.16: Thuật toán hàm paradoxicalNum sử dụng thuật toán Merge sort, sau khi thêm biến đếm số phép so sánh, số phép gán.

Ta sẽ cho bộ dữ liệu đầu vào **N ngẫu nhiên** trong phạm vi từ **0 - 500** và *các thành phần của mảng cũng được random ngẫu nhiên*, sẽ có được kết quả như sau:

N	*ARRAY*	*Assignments*		*Comparisons*	
		Thuật toán 1	***Thuật toán 2***	***Thuật toán 1***	***Thuật toán 2***
18	97, 361, 339, 492, 120, 237, 60, 233, 136, 15, 30, 50, 296, 173, 10, 454, 268, 303,	271	644	341	391
66	63, 72, 24, 5, 91, 47, 58, 89, 96, 34, 83, 34, 15, 93, 56, 79, 33, 89, 14, 90, 82, 65, 66, 87, 95, 43, 41, 22, 70, 58, 75, 31, 62, 32, 80, 94, 93, 80, 35, 53, 19, 44, 24, 54, 88, 69, 60, 28, 99, 91, 11, 88, 30, 54, 66, 62, 64, 51, 89, 46, 16, 8, 10, 30, 5, 60	3491	3066	4421	1922
144	361, 370, 63, 420, 280, 208, 103, 311, 43, 355, 79, 108, 339, 403, 279, 146, 333, 232, 478, 303, 366, 224, 29, 216, 166, 34, 145, 44, 129, 440, 162, 401, 304, 142, 399, 361, 48, 436, 252, 113, 50, 390, 210, 65, 186, 3, 466, 304, 471, 293, 88, 130, 91, 167, 401, 474, 58, 398, 111, 59, 355, 405, 174, 151, 18, 52, 487, 375, 410, 380, 232, 440, 91, 268, 151, 18, 489, 12, 73, 51, 445, 412, 78, 96, 374, 39, 259, 452, 310, 114, 441, 103, 245, 218, 317, 290, 489, 95, 279, 89, 288, 428,	15909	7632	20879	4883

	341, 307, 222, 251, 329, 137, 318, 15, 447, 169, 167, 133, 46, 72, 91, 51, 93, 32, 328, 234, 43, 332, 205, 24, 201, 120, 390, 361, 392, 313, 197, 85, 227, 361, 155, 300, 284, 60, 477, 38, 302, 239				
296	202, 444, 297, 220, 115, 80, 141, 195, 115, 344, 382, 155, 217, 292, 54, 422, 28, 498, 232, 479, 397, 161, 70, 108, 176, 33, 265, 143, 7, 395, 291, 456, 225, 326, 36, 279, 15, 471, 88, 16, 176, 312, 205, 262, 33, 438, 467, 377, 393, 466, 90, 113, 291, 321, 256, 455, 497, 246, 100, 124, 374, 365, 109, 317, 445, 266, 3, 445, 226, 210, 418, 275, 322, 354, 280, 113, 184, 43, 77, 13, 245, 269, 478, 182, 286, 361, 337, 488, 205, 142, 432, 338, 173, 100, 440, 485, 209, 348, 94, 360, 387, 500, 315, 394, 452, 212, 227, 407, 53, 207, 248, 37, 400, 294, 152, 389, 170, 465, 404, 224, 229, 271, 330, 311, 373, 50, 68, 273, 361, 294, 152, 69, 264, 316, 376, 186, 62, 327, 113, 254, 294, 89, 156, 271, 164, 109, 145, 200, 169, 400, 253, 96, 388, 445, 462, 491, 232, 9, 197, 400, 238, 287, 150, 35, 300, 293, 406, 330, 474, 265, 19, 199, 312, 383, 444, 117, 417, 247, 212, 284, 233, 152, 16, 136, 494, 140, 316, 221, 357, 400, 335, 441, 311, 214, 473, 210, 52, 243, 335, 72, 116, 246, 327, 220, 240, 102, 468, 109, 145, 436, 189, 60, 360, 327, 209, 75, 258, 47, 338, 214, 446, 271, 91, 108, 64, 74, 21, 28, 16, 452, 459, 383, 376, 38, 151, 490, 396, 208, 98, 476, 418, 302, 127, 275, 439, 258, 332, 396, 128, 59, 433, 19, 333, 326, 76, 431, 429, 175, 3, 391, 122, 190, 273, 78, 320, 471, 449, 2, 110, 49, 206, 69, 310, 280, 152, 213, 165, 121, 144, 179, 360, 385, 28, 202, 275, 102, 157, 215, 23, 99, 85, 29, 430, 144, 139, 119	67389	17393	87911	11257

Nhận xét:

Thuật toán 1:

+ Đếm số phép gán, số phép so sánh cho thuật toán:

Số phép gán:

+ bên ngoài vòng lặp có 2 phép gán cố định (1)
+ Vòng lặp ngoài có 2 phép gán cố định:
  + phép gán j = i + 1;
  + phép gán i = i + 1;

+ Vòng lặp sẽ chạy từ i = 0 đến i = n - 2
  ⇒ có n-1 phép gán => 2(n-1) phép gán ở vòng lặp ngoài (2)
+ Vòng lặp trong: Với mỗi i ∈ [0,n-2], vòng lặp trong sẽ chạy từ i+1 đến n-1
  ⇒ vòng lặp trong sẽ có n-i-1 lần chạy
  + Bên trong vòng lặp trong có 2 phép gán:
    + j = j+1 :luôn thực hiện qua mỗi vòng lặp => n-i-1 phép
    + count = count+1: thực hiện khi câu điều kiện arr[i] > arr[j] thỏa mãn

  Trong trường hợp tệ nhất (câu điều kiện arr[i] .> arr[j]) luôn thỏa ⇒ số lần thực hiện phép gán là: $\sum_i$ 2*(n-i-1) với i chạy từ 0 đến n-2 ⇒ có n(n-1) lần gán (3)

⇒ số phép gán là: (1) + (2) + (3)

⇔ 2 + n(n-1) + 2(n-1) = (n-1)(n+2) + 2 = **n^2 + n** (Trường hợp tệ nhất)

Số phép so sánh:

+ Vòng lặp ngoài:
  + Ta có, i sẽ chạy từ i = 0 đến i = n - 2 => sẽ có n-1 phép so sánh thỏa điều kiện vòng lặp và 1 phép so sánh không thỏa điều kiện (kết thúc vòng lặp) => n lần so sánh (1)
+ Vòng lặp trong:
  + với mỗi i ∈ [0,n-2], sẽ tương ứng mỗi j ∈ [i+1,n-1] => vòng lặp trong sẽ thực hiện
    + (n-i-1) phép so sánh thỏa điều kiện của while
    + (n-i-1) phép so sánh câu điều kiện if bên trong while.
    + 1 phép so sánh không thỏa điều kiện của while (kết thúc vòng lặp trong)

  => vòng lặp trong sẽ thực hiện $\sum_i$ 2*(n-i-1) + 1 với i ∈ [0,n-2] (2)

⇒ Số phép so sánh: (1) + (2)

⇔ n + n(n-1) + (n-1) = (n-1)*(n+1) + n = **n^2 + n - 1** (Trường hợp tệ nhất)

Từ số phép gán và số phép so sánh của **thuật toán 1**, ta có thể ước lượng độ phức tạp của thuật toán 1 là **O(n^2)**

Trường hợp tốt nhất: O(N)

Nếu mảng đã được sắp xếp tăng dần (không có vị trí thứ $A_i > A_j$ mà i<j

Trường hợp xấu nhất: O(N^2)

Nếu mảng được xếp giảm dần (tất cả các phần tử $A_i > A_j$ mà i < j

Trường hợp trung bình: O(N^2)

Mảng được xếp lộn xộn

Thuật toán 2:

Gọi T(n) là số phép gán của hàm và S(n) là số phép so sánh

Số phép gán:

+ Tại mỗi lần gọi đệ quy bên trong câu điều kiện, sẽ có 2 phép gán gọi đệ quy để cập nhật biến inv_count T(n/2) và T(n/2)
+ Phần còn lại là phép gán của hàm Merge để cập nhật biến inv_count, thao tác này tốn 1 phép gán qua mỗi lần gọi, và tốn tổng 1*n với n là số phép gán trong hàm merge.

Ta thấy hàm merge có tối đa an + b phép gán ⇒ công thức truy hồi số phép gán là: T(n) = T(n/2) + T(n/2) + an + b = 2T(n/2) + f(n) (1)

Số phép so sánh:

+ Tại mỗi bước kiểm tra điều kiện dừng, chỉ có một phép so sánh (left < right) ⇒ Công thức truy hồi của phép so sánh: S(n) = S(n/2) + 1 (2)

Từ (1) và (2), ta giải nghiệm theo định lý thợ, thu được độ phức tạp của hàm là **O(NlogN)**

⇒ Trường hợp tốt nhất = trường hợp xấu nhất = trường hợp trung bình = O(NlogN)

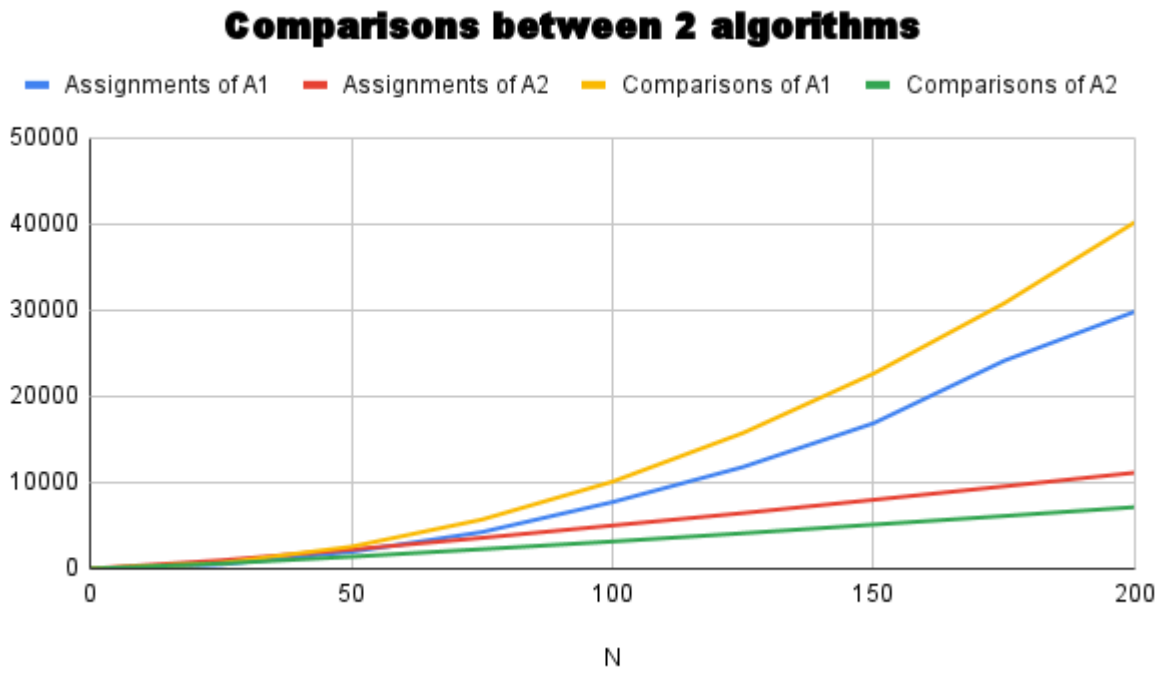
Trực quan hóa:

Bằng cách thêm các biến đếm số phép gán và số phép so sánh như ở trên, bây giờ ta sẽ cho ta sẽ cho bộ dữ liệu đầu vào N phạm vi từ 0 - 200 với bước nhảy 25 và các thành phần của mảng được random ngẫu nhiên, sẽ có được kết quả như sau:

(**Lưu ý:** *Vì các thành phần của mảng đều được random ngẫu nhiên nên qua mỗi lần chạy, số phép gán và phép so sánh của từng thuật toán sẽ có sự thay đổi, tuy nhiên ta so sánh độ lớn giữa 2 thuật toán, có thể dự đoán được thuật toán nào tối ưu hơn*)

N	*Assignments*		*Comparisons*	
	Assignments of A1	*Assignments of A2*	*Comparisons of A1*	*Comparisons of A2*
0	2	1	1	1
25	485	965	649	596
50	1945	2222	2549	1389
75	4245	3576	5699	2262
100	7728	5013	10099	3155
125	11802	6452	15749	4108
150	16882	7991	22649	5116
175	24134	9555	30799	6114
200	29828	11136	40199	7142

Bảng biểu 2.7: Số liệu trực quan hóa qua các số liệu.



Đồ thị 2.7: Sơ đồ trực quan hóa số liệu.

Nhận xét:

Ta thấy, số lượng phép gán và phép so sánh của thuật toán 1 có số lượng ngày càng lớn hơn nhiều so với số lượng phép gán và so sánh của thuật toán 2 khi N tăng dần

Nhận xét vì sao thuật toán tối ưu hơn thuật toán còn lại:

Ta có: Sự khác biệt giữa hai thuật toán A và B được gọi là đa thức (siêu đa thức) nếu tồn tại (với mọi) hằng số c sao cho $T_A(n) > n^c T_B(n)$ khi n đủ lớn. Khi đó A có độ phức tạp lớn hơn B với $T_A(n)$ và $T_B(n)$ là độ phức tạp thuật toán của A và B (Nguồn)

Thật vậy, ta thấy: $n^2 > n^{1/2}.\log n$ khi $n > 16$ => độ phức tạp của thuật toán 1 lớn hơn độ phức tạp của thuật toán 2 => thuật toán 2 tối ưu hơn thuật toán 1.

TÀI LIỆU THAM KHẢO

[1] *Merge sort* (2017) *Giải Thuật Lập Trình*. Available at: https://www.giaithuatlaptrinh.com/?tag=merge-sort (Accessed: 05 May 2024).

[2] (No date) *Chatgpt*. Available at: https://chat.openai.com/ (Accessed: 05 May 2024).