Lab 4: Sorting Algorithms

1 Lý thuyết về thuật toán sắp xếp

Bài toán sắp xếp (Sorting Problem):

- **Input:** Dãy n phần tử $\langle a_1, a_2, \ldots, a_n \rangle$
- **Output:** Một hoán vị $\langle a'_1, a'_2, \ldots, a'_n \rangle$ của dãy đầu vào sao cho $\langle a'_1 \leq a'_2 \leq \cdots \leq a'_n \rangle$

Phân loại thuật toán sắp xếp:

- **Comparison-based sorting**: Dựa trên phép so sánh giữa các phần tử.
  - **Cận dưới lý thuyết**: $\Omega(n \log n)$
  - **Ví dụ**: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort
- **Non-comparison sorting**: Không dựa trên phép so sánh.
  - Có thể đạt độ phức tạp tuyến tính $O(n)$ trong các điều kiện đặc biệt.
  - **Ví dụ**: Counting Sort, Radix Sort, Bucket Sort
- **In-place sorting**: Sử dụng bộ nhớ phụ $O(1)$ hoặc $O(\log n)$.
- **Stable sorting**: Giữ nguyên thứ tự tương đối của các phần tử bằng nhau.

1.1

Bảng 1

Algorithm	Best Case	Average Case	Worst Case	Stable	In-place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Có	Có
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Không	Có
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Có	Có
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Có	Không
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Không	Có

2 Các thuật toán sắp xếp cơ bản

2.1 Bubble Sort

Ý tưởng: So sánh và hoán đổi các cặp phần tử liên tiếp, đưa phần tử lớn nhất "nổi"lên cuối mảng sau mỗi lượt.

Yêu cầu:

a. Viết *pseudocode* cho thuật toán Bubble Sort.

b. Cài đặt thuật toán Bubble Sort trong C++.

c. Tối ưu hóa thuật toán bằng cách dừng sớm khi mảng đã được sắp xếp.

d. So sánh số phép so sánh và số phép hoán đổi giữa phiên bản cơ bản và phiên bản tối ưu với các bộ test khác nhau.

Test cases gợi ý: Mảng đã sắp xếp xuôi/ngược, mảng ngẫu nhiên, mảng có phần tử trùng,...

Bài làm

a. Viết *pseudocode* cho thuật toán Bubble Sort.

BUBBLE-SORT(A)
1 $n = A.lenght$
2 **for** $i = 0$ **to** $n - 1$
3 **for** $j = 0$ **to** $n - i - 1$
4 **if** $A[j] > A[j + 1]$
5 SWAP($A[j], A[j + 1]$)
6 $swapped =$ TRUE

b. Cài đặt thuật toán Bubble Sort trong C++.

```
void bubble_sort(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (a[j] > a[j + 1]) {
                swap(a[j], a[j + 1]);
                swapped = true;
            }
        }
    }
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

c. Tối ưu hóa thuật toán bằng cách dừng sớm khi mảng đã được sắp xếp.

```
BUBBLE-SORT-OPTIM(A)
1  n = A.length
2  for i = 0 to n − 1
3      swapped = FALSE
4      for j = 0 to n − i − 1
5          if A[j] > A[j + 1]
6              SWAP(A[j], A[j + 1])
7              swapped = TRUE
8      if ¬swapped
9          return
```

d. So sánh số phép so sánh và số phép hoán đổi giữa phiên bản cơ bản và phiên bản tối ưu với các bộ test khác nhau.

- Như bên trên.

2.2 Selection Sort

Ý tưởng: Tìm và đưa phần tử nhỏ nhất trong phần chưa sắp xếp và đặt nó vào vị trí đầu của phần chưa sắp xếp đó.

Yêu cầu:

a. Viết pseudocode cho thuật toán Selection Sort.

b. Cài đặt thuật toán Selection Sort trong C++.

c. Giải thích tại sao Best case và Worst case có cùng độ phức tạp.

d. Chứng minh rằng Selection Sort thực hiện đúng $n - 1$ lần hoán đổi (là số lần hoán đổi tối thiểu).

Bài làm

a. Viết *pseudocode* cho thuật toán Selection Sort.

```
SELECTION-SORT(A)
1  n = A.length
2  for i = 0 to n − 1
3      min_index = i
4      for j = i + 1 to n − 1
5          if A[j] < A[min_index]
6              min_index = j
7      SWAP(A[I], A[MIN_INDEX])
```

b. Cài đặt thuật toán Selection Sort trong C++.

```cpp
void selection_sort(int *a, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min_idx]) {
                min_idx = j;
            }
        }
        swap(a[i], a[min_idx]);
    }
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

c. Giải thích tại sao Best case và Worst case có cùng độ phức tạp.
Trong thuật toán Selection Sort, tại mỗi vòng lặp, ta phải tìm phần tử nhỏ nhất trong phần chưa sắp xếp bằng cách duyệt qua toàn bộ các phần tử còn lại. Việc này được thực hiện độc lập với việc mảng ban đầu có được sắp xếp hay không. Do đó, số lần so sánh luôn là:

$$\frac{n(n-1)}{2}$$

ở cả trường hợp tốt nhất và xấu nhất.

Vì vậy:

$$T_{best}(n) = T_{worst}(n) = O(n^2)$$

d. Chứng minh rằng Selection Sort thực hiện đúng $n-1$ lần hoán đổi
Ở mỗi vòng lặp i (từ 1 đến $n-1$), thuật toán tìm được vị trí phần tử nhỏ nhất trong phần còn lại và nếu vị trí đó khác i, thì hoán đổi hai phần tử. Như vậy, mỗi vòng lặp ngoài thực hiện tối đa một lần hoán đổi.

Vì có $(n-1)$ vòng lặp ngoài, nên số lần hoán đổi tối đa là $(n-1)$. Trên thực tế, thuật toán luôn thực hiện đúng $(n-1)$ lần hoán đổi, vì ở vòng cuối cùng, phần tử còn lại đã nằm đúng vị trí.

2.3 Insertion Sort

Ý tưởng: Tìm và đưa phần tử nhỏ nhất trong phần chưa sắp xếp và đặt nó vào vị trí đầu của phần chưa sắp xếp đó. **Yêu cầu:**

a. Viết pseudocode cho thuật toán Insertion Sort.

b. Cài đặt thuật toán Insertion Sort trong C++.

c. Cài đặt phiên bản Binary Insertion Sort (sử dụng binary search để tìm vị trí chèn). Phân tích độ phức tạp và so sánh với phiên bản thường.

d. Giải thích tại sao Insertion Sort hiệu quả với Mảng nhỏ (n $<$ 50) và mảng gần như đã sắp xếp.

Bài làm

a. Viết pseudocode cho thuật toán Insertion Sort.

INSERTION-SORT(A)

```
1  n = A.length
2  for i = 1 to n − 1
3      key = A[i]
4      j = i − 1
5      while (j ≥ 0)  &  (A[j] > key)
6          A[j + 1] = a[j]
7          j = j − 1
8      A[j + 1] = key
```

b. Cài đặt thuật toán Insertion Sort trong C++.

```cpp
void insertion_sort(int *a, int n) {
    for (int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = key;
    }
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

c. Cài đặt phiên bản Binary Insertion Sort (sử dụng binary search để tìm vị trí chèn). Phân tích độ phức tạp và so sánh với phiên bản thường.

```
int binary_search(int *a, int key, int l, int r) {
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (a[mid] == key) {
            return mid;
        } else if (a[mid] < key) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return l;
}

void binary_insertion_sort(int *a, int n) {
    for (int i = 1; i < n; ++i) {
        int key = a[i];
        int pos = binary_search(a, key, 0, i - 1);
        int j = i - 1;
        while (j >= pos) {
            a[j + 1] = a[j];
            j--;
        }
        a[pos] = key;
    }
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

d. Giải thích tại sao Insertion Sort hiệu quả với Mảng nhỏ (n $<$ 50) và mảng gần như đã sắp xếp.

- **Với mảng nhỏ ($n < 50$):**
  - **Overhead (Chi phí) thấp:** Thuật toán cực kỳ đơn giản (vài vòng lặp, không đệ quy, không cần bộ nhớ phụ $O(1)$).
  - Các thuật toán $O(n \log n)$ (như QuickSort) cồng kềnh hơn, tốn chi phí cho việc gọi hàm, quản lý đệ quy.

  - Khi n nhỏ, n^2 vẫn là một con số nhỏ, và sự đơn giản của Insertion Sort chạy nhanh hơn sự cồng kềnh của các thuật toán phức tạp.
- **Với mảng gần như đã sắp xếp:**
  - **Tính thích ứng (Best Case $O(n)$):** Khi một phần tử đã đúng vị trí (xảy ra thường xuyên), Insertion Sort chỉ mất **1 phép so sánh** (tức $O(1)$).
  - Nó không tốn thời gian di dời cho những phần tử đã đúng chỗ.
  - Kết quả là tổng thời gian chạy tiến rất gần về $O(n)$, nhanh hơn cả các thuật toán $O(n \log n)$.

3 Các thuật toán sắp xếp hiệu quả

3.1 Merge Sort

Ý tưởng: Sử dụng kỹ thuật chia để trị (Divide and Conquer):

- **Chia (Divide):** Chia mảng thành 2 nửa.
- **Trị (Conquer):** Sắp xếp đệ quy từng nửa.
- **Hợp nhất (Combine):** Trộn 2 mảng con đã sắp xếp.

Yêu cầu:

a. Viết pseudocode cho thuật toán Merge Sort và hàm Merge.

b. Thiết lập công thức truy hồi và sử dụng Master Theorem để phân tích độ phức tạp thời gian.

c. Cài đặt Merge Sort trong C++ với cả hai cách:

- Top-down (đệ quy)
- Bottom-up (lặp)

d. Chứng minh Merge Sort là thuật toán ổn định (stable).

e. *Phân tích tại sao Merge Sort cần $O(n)$ bộ nhớ phụ?*

Bài làm

a. Viết pseudocode cho thuật toán Merge Sort và hàm Merge.

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5        L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7        R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13        if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16        else A[k] = R[j]
17              j = j + 1
```

```
MERGE-SORT(A, p, r)
1  if p < r
2        q = ⌊(p + r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, p + 1, r)
5        MERGE(A, p, q, r)
```

b. Thiết lập công thức truy hồi và sử dụng Master Theorem để phân tích độ phức tạp thời gian.

$$
\begin{aligned}
&T(n) = 2T(\frac{n}{2}) + O(n) \\
&a = 2, \quad b = 2, \quad f(n) = n \\
&n^{\log_b a} = n = f(n) \\
\Rightarrow &T(n) = \Theta(n \log(n))
\end{aligned}
$$

c. Cài đặt Merge Sort trong C++ với cả hai cách:

- Top-down (đệ quy)

```
void merge(int *L, int l_size, int *R, int r_size, int *a) {
    int i = 0, l = 0, r = 0;
    while (l < l_size && r < r_size) {
        if (L[l] <  R[r]){
            a[i] = L[l];
            i++; l++;
        }
        else {
            a[i] = R[r];
            r++; i++;
        }
    }
    while (l < l_size) {
        a[i] = L[l];
        i++; l++;
    }
    while (r < r_size) {
        a[i] = R[r];
        i++; r++;
    }
}
void merge_sort(int *a, int n) {
    if (n <= 1) return;
    int middle = n/2;
    int i = 0, j = 0;
    int L[middle], R[n - middle];
    for (i = 0; i < n; i++) {
        if (i < middle){
            L[i] = a[i];
        }
        else {
            R[j] = a[i];
            j++;
        }
    }
    merge_sort(L, middle);
    merge_sort(R, n - middle);
    merge(L, middle, R, n - middle, a);
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

- Bottom-up (lặp)

```cpp
void bottom_up_merge_sort(int *a, int n) {
    int *temp = new int[n];

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

            int i = left, j = mid + 1, k = left;
            while (i <= mid && j <= right) {
                temp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
            }
            while (i <= mid) temp[k++] = a[i++];
            while (j <= right) temp[k++] = a[j++];

            for (int p = left; p <= right; ++p)
                a[p] = temp[p];
        }
    }
    delete temp;
}
```

```
array: 37 23 0 17 12 72 31 46 100 88 54
sorted: 0 12 17 23 31 37 46 54 72 88 100
```

d. Chứng minh Merge Sort là thuật toán ổn định (stable).
Merge Sort **ổn định** vì hàm `Merge` được thiết kế có ưu tiên. Khi trộn hai mảng con L (trái) và R (phải), nếu gặp hai phần tử bằng nhau ($L[i] = R[j]$), thuật toán **luôn chọn phần tử từ mảng L (bên trái) trước**. Điều này đảm bảo thứ tự tương đối ban đầu của các phần tử bằng nhau được bảo toàn.

e. *Phân tích tại sao Merge Sort cần $O(n)$ bộ nhớ phụ?*
Nó cần $O(n)$ bộ nhớ phụ để chứa các **mảng tạm (L và R)** trong quá trình `Merge`.

3.2 Quick Sort

Ý tưởng: Sử dụng kỹ thuật chia để trị với phương pháp partition::

- **Chọn pivot:** Chọn một phần tử làm pivot
- **Partition:** Chia mảng thành 2 phần: nhỏ hơn pivot và lớn hơn pivot
- **Đệ quy:** Sắp xếp đệ quy hai phần

Yêu cầu:

a. Viết pseudocode cho thuật toán Quick Sort và hàm Partition.

b. Cài đặt Quick Sort với các phương pháp chọn pivot khác nhau (đầu mảng, cuối mảng, ngẫu nhiên).

c. So sánh hiệu quả của từng phương pháp chọn pivot với:

- Mảng ngẫu nhiên
- Mảng đã sắp xếp
- Mảng có nhiều phần tử trùng

Bài làm

a. Viết pseudocode cho thuật toán Quick Sort và hàm Partition.

```
PARTITION(A, l, r)
 1  x = A[⌊(l + r)/2⌋]
 2  i = l − 1
 3  j = r + 1
 4  while 1
 5       repeat
 6            j = j − 1
 7       until A[j] ≤ x
 8       repeat
 9            i = i + 1
10       until A[i] ≥ x
11       if i < j
12            SWAP(A[i], A[j])
13       else
14            return j
```

```
QUICKSORT(A, l, r)
1  if l < r
2       q = PARTITION(A, l, r)
3       QUICKSORT(A, l, q)
4       QUICKSORT(A, q + 1, r)
```

b. Cài đặt Quick Sort với các phương pháp chọn pivot khác nhau (đầu mảng, cuối mảng, ngẫu nhiên).

```
int hoare_partition(int *a, int l, int r) {
    int x = a[(l + r)/2];
    int i = l - 1;
    int j = r + 1;

    while (1) {
        do  j--;
        while (a[j] > x);
        do i++;
        while (a[i] < x);

        if (i < j) {
            swap(a[i], a[j]);
        } else return j;
    }
}
int lomuto_partition(int *a, int l, int r) {
    int pivot = a[r];
    int i = l - 1;
    for (int j = l; j < r; j++) {
        if (a[j] < pivot) {
            i++;
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[r]);
    return i + 1;
}
void quick_sort(int *a, int l, int r) {
    if (l < r) {
        int q = hoare_partition(a, l, r);
        quick_sort(a, l, q);
        quick_sort(a, q + 1, r);

        // int q = lomuto_partition(a, l, r);
        // quick_sort(a, l, q - 1);
        // quick_sort(a, q + 1, r);
    }
}
```

```
array: 64, -10, 8, 15, 27, 8, 5, -2, 99, 31
sorted: -10, -2, 5, 8, 8, 15, 27, 31, 64, 99
```

c. So sánh hiệu quả của từng phương pháp chọn pivot với:

- Mảng ngẫu nhiên
- Mảng đã sắp xếp
- Mảng có nhiều phần tử trùng