VIETNAM NATIONAL UNIVERSITY - HO CHI MINH

UNIVERSITY OF SCIENCE

Sorting Algorithms & Performance Study

Techincal Report

CSC10004 - Data Structure And Algorithm

Students:

24120059 - Trần Kim Hữu

24120041 - Phạm Võ Đức

24120006 - Đào Thanh Phong

24120069 - Trần Hoài Bảo Khang

Supervisors:

Lect. Lê Nhựt Nam

Ngày 26 tháng 4 năm 2025

LỜI CẢM ƠN

Với sự phát triển bùng nổ của công nghệ thông tin trong thời đại hiện nay, số lượng dữ liệu người dùng ngày một gia tăng nhanh chóng, dẫn đến sự quan tâm về phát triển của các thuật toán sắp xếp cũng gia tăng. Điều này thu hút một lượng lớn sự quan tâm từ các nhà nghiên cứu về việc phát triển các thuật toán sắp xếp. Từ đó rất nhiều thuật toán sắp xếp đã ra đời với những ưu và nhược điểm điểm riêng. Đề tài "Sorting Algorithms & Performance Study" được hoàn thành với sự hướng dẫn của giảng viên Lê Nhựt Nam. Qua đây chúng em xin chân thành cảm ơn thầy đã giúp chúng em hoàn thành đề tài này.

MỤC LỤC

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ

TÓM TẮT

Tìm hiểu về các 13 thuật toán sắp xếp phổ biến: Selection sort, Insertion sort, Binary Insertion sort, Bubble sort, Shaker sort, Shell sort, Heap sort, Merge sort, Natural Merge sort, Quick sort, std::sort, Radix sort, Counting sort. So sánh hiệu năng (thời gian thực thi thực tế) của các thuật toán trên dựa trên tiêu chí về bộ dữ liệu, kích thước bộ dữ liệu. Từ kết quả thống kê và phân tích, chúng ta có thể xác định từng thuật toán sẽ phù hợp đối với bộ dữ liệu như thế nào.

Đối với bộ dữ liệu số nguyên có độ lớn nhỏ hơn 20000 phần tử, các thuật toán cho thấy hiệu năng thực thi gần như tương đương nhau, các thuật toán sắp xếp cơ bản ($\mathcal{O}(n^2)$) như Bubble Sort, Selection Sort, Insertion Sort,.... vẫn có hiệu quả. Nhưng đối với những bộ dữ liệu có số phần tử lớn hơn, các thuật toán sắp xếp cấp cao ($\mathcal{O}(n \log n)$) như Quick Sort, Merge Sort, Heap Sort, std::sort,... thể hiện hiệu quả rõ rệt hơn, trong khi các thuật toán sắp xếp cơ bản không còn phù hợp. Radix Sort, một thuật toán sắp xếp đặc biệt ($\mathcal{O}(kn)$), cũng thể hiện một hiệu năng khá mạnh mẽ vì không phụ thuộc quá nhiều vào số lượng phần tử mà phụ thuộc phần lớn vào số chữ số của phần tử lớn nhất trong mảng. Tương tự, Counting Sort ($\mathcal{O}(kn)$) cho ra một hiệu năng thực thi ổn định do phụ thuộc chủ yếu vào độ chênh lệch giữa phần tử lớn nhất và nhỏ nhất.

Đối với bộ dữ liệu là chuỗi (ở đây là dạng từ điển), các thuật toán sắp xếp cấp cao vẫn cho thấy hiệu quả hơn hẳn so với các thuật toán sắp xếp cơ bản. Trong khi đó, Radix Sort và Counting Sort lại có hiệu năng thực thi kém nhất trong số 13 thuật toán được dùng để khảo sát.

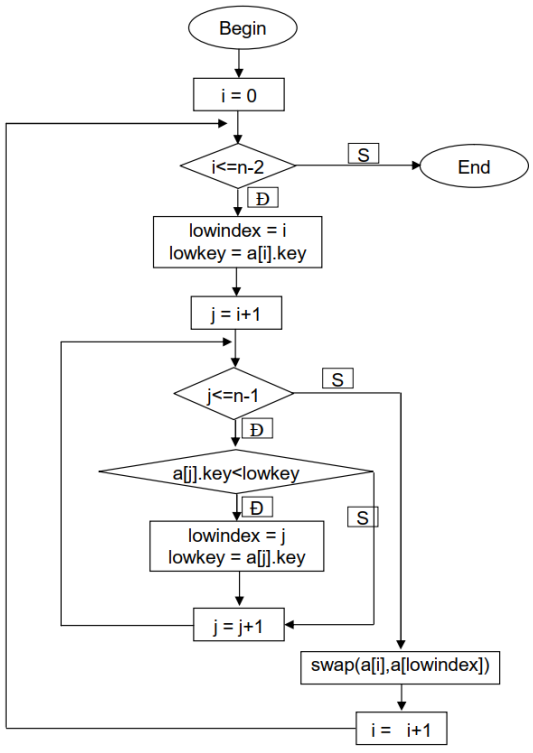
KHẢO SÁT CÁC THUẬT TOÁN

1. Selection sort

1.1. Ý tưởng thuật toán

- Tìm phần tử nhỏ nhất đưa vào ví trí index 0.
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí index 1.
- Tìm phần tử nhỏ tiếp theo đưa vào vị trí index 2.
- ...

1.2. Lưu đồ thuật toán



Hình 1.1: Lưu đồ Selection sort

1.3. Code tham khảo

```
void selectionSort(vector<int>& a) {
    for (int i = 0; i < a.size() - 1; i++) {
        int pos = i;
        for (int j = i + 1; j < a.size(); j++) {
            if (a[j] < a[pos]) {
                pos = j;
            }
        }
        swap(a[pos], a[i]);
    }
}
```

1.4. Phân tích thuật toán

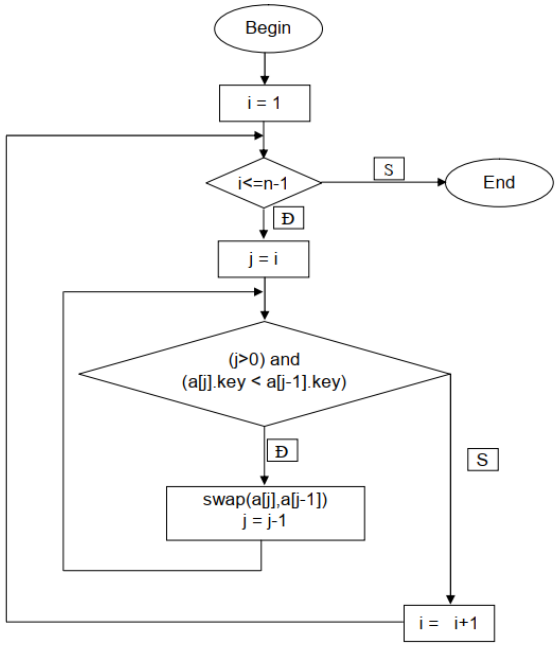
- Best case: khi dãy đầu vào đã được sắp xếp, 0 đổi chỗ, khoảng $n^2/2$ so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Worst case: Khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp, $n-1$ đổi chỗ và khoảng $n^2/2$ so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Average case: $\mathcal{O}(n)$ đổi chỗ và khoảng $n^2/2$ so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Ưu điểm nổi bật của Selection sort là số phép đổi chỗ ít. Điều này có ý nghĩa nếu như việc đổi chỗ là tốn kém.

2. Insertion sort

2.1. Ý tưởng thuật toán

- Tại bước k = 1, 2, ..., n, đưa phần tử thứ k trong mảng đã cho vào đúng vị trí trong dãy gồm k phần tử đầu tiên.
- Kết quả là sau bước k, k phần tử đầu tiên được sắp theo thứ tự.

2.2. Lưu đồ thuật toán



Hình 1.2: Lưu đồ Insertion sort

2.3. Code tham khảo

```
void insertionSort(vector<int>& a) {
    for (int i = 1; i < a.size(); i++) {
        int temp = a[i];
        int j = i - 1;
        while (j >= 0 && a[j] > temp) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = temp;
    }
}
```

2.4. Phân tích thuật toán

- Best case: khi dãy đầu vào đã được sắp xếp, 0 đổi chỗ, khoảng $n-1$ so sánh. Độ phức tạp: $\mathcal{O}(n)$.
- Worst case: Khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp, khoảng $n^2/2$ đổi chỗ và so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Average case: khoảng $n^2/4$ đổi chỗ và so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Là thuật toán sắp xếp tốt đối với dãy đã gần được sắp xếp, nghĩa là mỗi phần tử đã đứng ở vị trí rất gần vị trí trong thứ tự cần sắp xếp.

3. Binary Insertion sort

3.1. Ý tưởng thuật toán

Giống như Insertion sort nhưng thay vì tìm vị trí chèn bằng cách duyệt tuần tự, ta dùng Binary Search để tìm nhanh vị trí chèn

3.2. Code tham khảo

```
int binarySearch(vector<int>& a, int left, int right, int key) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (a[mid] > key) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
    return left;
}
void binaryInsertionSort(vector<int>& a) {
    for (int i = 1; i < a.size(); i++) {
```

```
15          int key = a[i];
16          int pos = binarySearch(a, 0, i - 1, key);
17          for (int j = i; j > pos; j--) {
18              a[j] = a[j - 1];
19          }
20          a[pos] = key;
21      }
22  }
```

3.3. Phân tích thuật toán

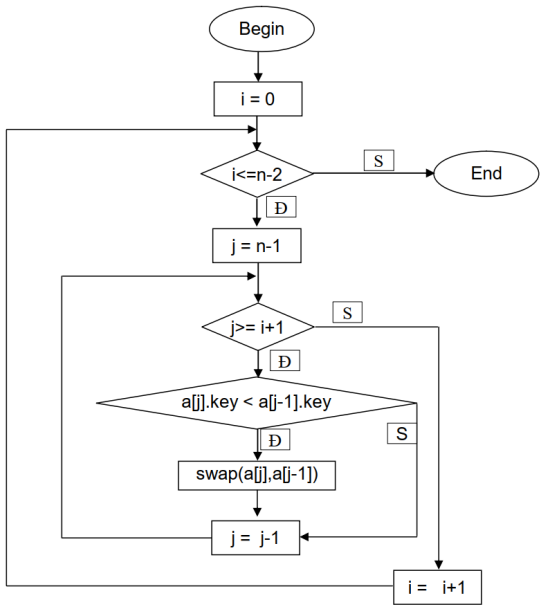
- Giảm số lần so sánh từ $\mathcal{O}(n)$ xuống $\mathcal{O}(\log n)$ cho mỗi phần tử.
- Tổng số phép so sánh giảm từ $\mathcal{O}(n^2)$ xuống $\mathcal{O}(n \log n)$
- Vẫn giữ ưu điểm của Insertion Sort với dữ liệu nhỏ hoặc gần sắp xếp nhưng phức tạp hơn Insertion sort.
- Vẫn cần $\mathcal{O}(n^2)$ phép dịch chuyển phần tử trong trường hợp xấu nhất.

4. Bubble sort

4.1. Ý tưởng thuật toán

- Duyệt từ cuối dãy về đầu dãy: Bắt đầu từ phần tử cuối cùng (j = a.size() - 1), so sánh với phần tử liền trước nó (j-1).
- Nếu a[j-1] > a[j], thực hiện hoán đổi hai phần tử này.
- Sau mỗi lần duyệt phần tử nhỏ nhất sẽ "nổi lên" vị trí i.

4.2. Lưu đồ thuật toán



Hình 1.3: Lưu đồ Bubble sort

4.3. Code tham khảo

```
void bubbleSort(vector<int>& a) {
    for (int i = 0; i <= a.size() - 2; i++) {
        for (int j = a.size() - 1; j >= i + 1; j--) {
            if (a[j - 1] > a[j]) {
                swap(a[j - 1], a[j]);
            }
        }
    }
}
```

4.4. Phân tích thuật toán

- Best case: khi dãy đầu vào đã được sắp xếp, 0 đổi chỗ, khoảng $n^2/2$ so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.

- Worst case: Khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp, khoảng $n^2/2$ đổi chỗ và so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Average case: khoảng $n^2/4$ đổi chỗ và khoảng $n^2/2$ so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Tốn nhiều phép so sánh và hoán đổi ngay cả khi mảng đã gần đúng thứ tự.

5. Shaker sort

5.1. Ý tưởng thuật toán

- Duyệt xuôi: so sánh và hoán đổi các cặp phần tử liền kề từ $left$ đến $right$, sau lượt duyệt này, phần tử lớn nhất sẽ nằm ở $right$. Giảm $right$.
- Duyệt ngược: so sánh và hoán đổi các cặp phần tử liền kề từ $right$ đến $left$, sau lượt duyệt này, phần tử nhỏ nhất sẽ nằm ở $left$. Tăng $left$.
- Lặp lại cho đến khi $left \geq right$.

5.2. Code tham khảo

```
void shakerSort(vector<int>& a) {
    int left = 0, right = a.size() - 1;
    while (left < right) {
        for (int i = left; i < right; i++) {
            if (a[i + 1] < a[i]) {
                swap(a[i], a[i + 1]);
            }
        }
        right--;
        for (int i = right; i > left; i--) {
            if (a[i] < a[i - 1]) {
                swap(a[i], a[i - 1]);
            }
        }
        left++;
```

```
    }
}
```

5.3. Phân tích thuật toán

- Là một phiên bản cải tiến của Bubble sort với khả năng sắp xếp hiệu quả hơn bằng cách duyệt mảng theo cả hai chiều.
- Giảm khoảng 25-50% số phép so sánh so với Bubble sort.

6. Shell sort

6.1. Ý tưởng thuật toán

- Chọn $gap = n/2$.
- Với mỗi *gap*, chia mảng thành các sub-array cách nhau *gap* phần tử, thực hiện sắp xếp từng sub-aray bằng Insertion sort.
- Giảm $gap = gap/2$ cho đến khi $gap = 1$.

6.2. Code tham khảo

```
void shellSort(vector<int>& a) {
    for (int gap = a.size() / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.size(); i++) {
            int temp = a[i];
            int j = i - gap;
            while (j >= 0 && a[j] > temp) {
                a[j + gap] = a[j];
                j -= gap;
            }
            a[j + gap] = temp;
        }
    }
}
```

6.3. Phân tích thuật toán

- Là một cải tiến của Insertion sort, hiệu năng phụ thuộc vào cách chọn *gap*.
- Best case: khi dãy đầu vào đã được sắp xếp, mỗi lượt với *gap* khác nhau chỉ cần duyệt qua mảng 1 lần, số lượt duyệt là $\log_2(n)$. Độ phức tạp: $\mathcal{O}(n \log n$.
- Worst case: Khi dãy đầu vào có thứ tự ngược lại với thứ tự cần sắp xếp, khoảng $n^2/4$ phép so sánh. Độ phức tạp: $\mathcal{O}(n^2)$.
- Average case: Độ phức tạp khoảng: $\mathcal{O}(n^{3/2})$.

7. Heap sort

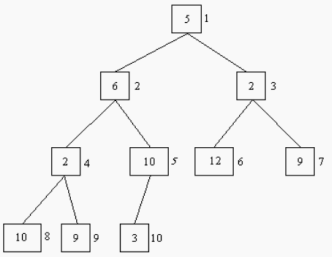
7.1. Ý tưởng thuật toán

- Xây dựng cây Heap: sử dụng thao tác hiệu chỉnh Heapify đê chuyển một mảng bình thường thành một mảng Heap.
- Thao tác sắp xếp:
    1. Hoán vị phần tử cuối cùng của mảng Heap với phần tử đầu tiên của Heap.
    2. Loại bỏ phần tử cuối cùng khỏi mảng Heap.
    3. Thực hiện thao tác hiệu chỉnh Heapify với phần còn lại của mảng Heap.
- Lưu ý:
    1. Tất cả các phần tử trên mảng Heap có chỉ số [n/2] đến [n-1] đều là nút lá.
    2. Thực hiện thao tác hiệu chỉnh Heapify cho các phần tử có chỉ số từ [n/2-1] đến [0].

7.2. Ví dụ thuật toán

Khoá	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
Ban đầu	5	6	2	2	10	12	9	10	9	3

Mảng này được xem như là một cây nhị phân ban đầu như sau:

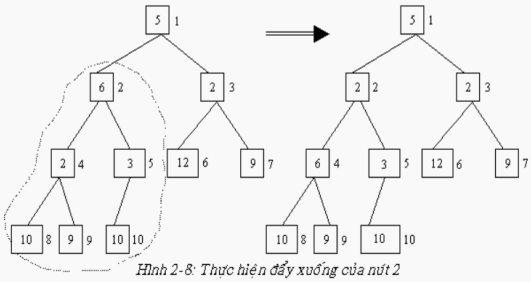


Hình 2-6: Cây ban đầu

Trong cây trên, giá trị ghi trong các nút là khoá của các phần tử mảng, giá trị ghi bên ngoài các nút là chỉ số của các phần tử mảng.

Hình 1.4: Step 1

Nút 4 và nút 3 đã đúng vị trí nên không phải thực hiện sự hoán đổi. Tại nút 2, giá trị khóa của nó lớn hơn khoá con trái và khoá của con trái nhỏ hơn khoá của con phải nên ta hóan đổi nút 2 cho con trái của nó (nút 4), sau khi hoán đổi, ta xét lại nút 4, thấy nó vẫn đúng vị trí nên kết thúc việc đẩy xuống của nút 2.

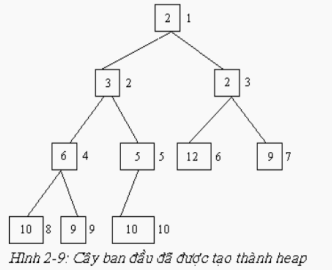


Hình 2-8: Thực hiện đẩy xuống của nút 2

Cuối cùng ta xét nút 1, ta thấy giá trị khoá của nút 1 lớn hơn khoá của con trái và con trái có khoá bằng khoá của con phải nên hóan đổi nút 1 cho con trái của nó (nút 2).

Hình 1.5: Step 3

Sau khi thực hiện phép hóan đổi nút 1 cho nút 2, ta thấy nút 2 có giá trị khoá lớn hơn khoá của con phải của nó (nút 5) và con phải có khoá nhỏ hơn khoá của con trái nên phải thực hiện phép hoán đổi nút 2 cho nút 5. Xét lại nút 5 thì nó vẫn đúng vị trí nên ta được cây mới trong hình 2-9.

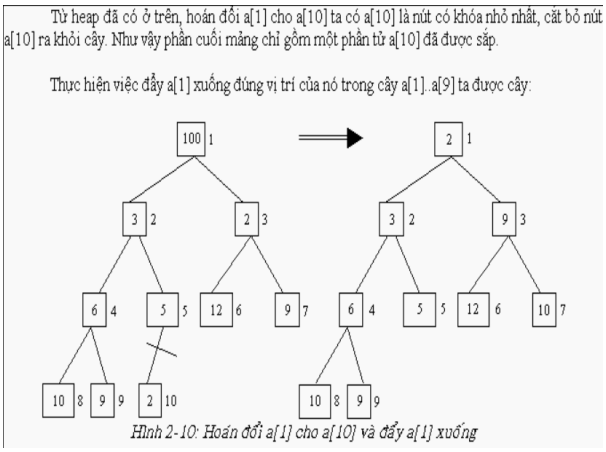


Hình 2-9: Cây ban đầu đã được tạo thành heap

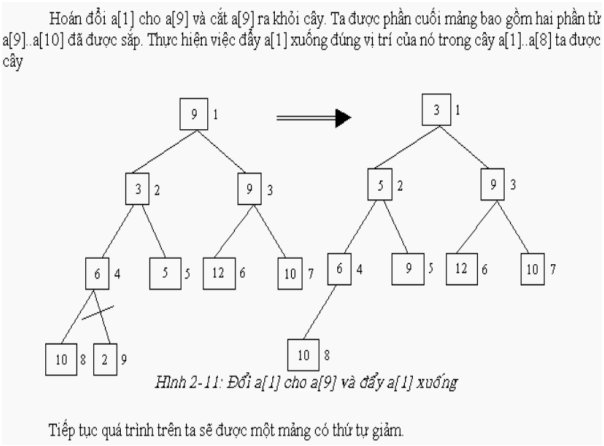
Cây này là một heap tương ứng với mảng

Khoá	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
Heap	2	3	2	6	5	12	9	10	9	10

Hình 1.6: Step 4



Hình 1.7: Step 5



Hình 1.8: Step 6

7.3. Code tham khảo

```
void heapify(vector<int>& a, int n, int i) {
    int saved = a[i];
    while (i < n / 2) {
        int child = i * 2 + 1;
        if (child + 1 < n) {
            if (a[child] < a[child + 1]) {
                child++;
            }
        }
        if (saved >= a[child]) break;
```

```
            a[i] = a[child];
            i = child;
        }
}
void buildHeap(vector<int>& a, int n) {
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(a, n, i);
        }
}
void heapSort(vector<int>& a) {
        buildHeap(a, a.size());
        for (int i = a.size() - 1; i > 0;i--) {
            swap(a[i], a[0]);
            heapify(a, i, 0);
        }
}
```

7.4. Phân tích thuật toán

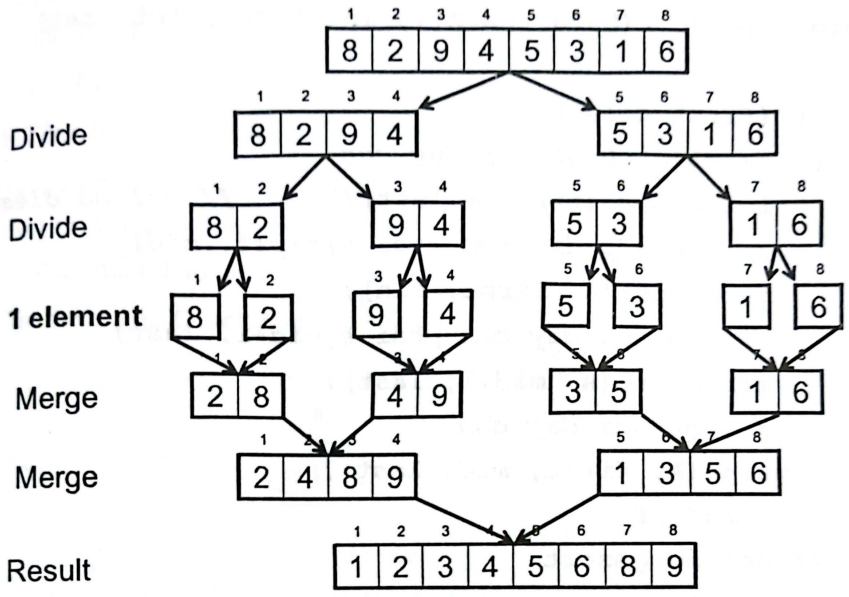
- Độ phức tạp ổn định $\mathcal{O}(n \log n)$ trong mọi trường hợp.
- Hiệu quả với dữ liệu lớn.
- Không đệ quy, tránh tràn stack.

8. Merge sort

8.1. Ý tưởng thuật toán

- Chia dãy gồm n phần tử cần sắp xếp ra thành 2 dãy mỗi dãy có n/2 phần tử.
- Sắp xếp mỗi dãy con bằng đệ quy sử dụng Merge sort. Khi dãy chỉ còn một phần tử thì trả lại phần tử này.
- Trộn hai dãy con được sắp xếp để thu được dãy đã sắp xếp gồm tất cả các phần tử của cả hai dãy con.

8.2. Ví dụ thuật toán



Hình 1.9: Merge Sort

8.3. Code tham khảo

```
void merge(vector<int>& a, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int>L1(n1), L2(n2);
    for (int i = 0; i < n1; i++) {
        L1[i] = a[i + left];
    }
    for (int i = 0; i < n2; i++) {
        L2[i] = a[i + mid + 1];
    }
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L1[i] <= L2[j]) {
            a[k] = L1[i];
            i++;
```

```
16          }
17          else {
18              a[k] = L2[j];
19              j++;
20          }
21          k++;
22      }
23      while (i < n1) {
24          a[k] = L1[i];
25          k++;
26          i++;
27      }
28      while (j < n2) {
29          a[k] = L2[j];
30          k++;
31          j++;
32      }
33  }
34  void mergeSort(vector<int>& a, int left, int right) {
35      if (left < right) {
36          int mid = left + (right - left) / 2;
37          mergeSort(a, left, mid);
38          mergeSort(a, mid + 1, right);
39          merge(a, left, mid, right);
40      }
41  }
```

8.4. Phân tích thuật toán

- Độ phức tạp ổn định $\mathcal{O}(n \log n)$ trong mọi trường hợp.
- Merge Sort là một thuật toán ổn định, có nghĩa là nếu hai phần tử có giá trị bằng nhau, thứ tự của chúng sẽ không bị thay đổi sau khi sắp xếp.

9. Matural Merge sort

9.1. Ý tưởng thuật toán

Natural Merge sort là biến thể của Merge sort tận dụng các dãy con đã sắp xếp trước (runs) trong mảng thay vì chia đều như Merge sort.

9.2. Code tham khảo

```
void merge(vector<int>& a, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int>L1(n1), L2(n2);
    for (int i = 0; i < n1; i++) {
        L1[i] = a[i + left];
    }
    for (int i = 0; i < n2; i++) {
        L2[i] = a[i + mid + 1];
    }
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L1[i] <= L2[j]) {
            a[k] = L1[i];
            i++;
        }
        else {
            a[k] = L2[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        a[k] = L1[i];
        k++;
```

```
        i++;
    }
    while (j < n2) {
        a[k] = L2[j];
        k++;
        j++;
    }
}
void naturalMergeSort(vector<int>& a) {
    if (a.size() <= 1) return;
    while (true) {
        vector<pair<int, int>>runs;
        int start = 0;
        while (start < a.size()) {
            int end = start;
            while (end + 1 < a.size() && a[end] <= a[end + 1]) {
                end++;
            }
            runs.push_back({ start, end });
            start = end + 1;
        }
        if (runs.size() <= 1) return;
        for (int i = 0; i + 1 < runs.size(); i += 2) {
            merge(a, runs[i].first, runs[i].second, runs[i + 1].second);
        }
    }
}
```

9.3. Phân tích thuật toán

- Best case: mảng đã sắp xếp (chỉ có 1 runs), độ phức tạp $\mathcal{O}(n)$.
- Wost case: mảng đã sắp xếp ngược (n runs), độ phức tạp $\mathcal{O}(n \log n)$.

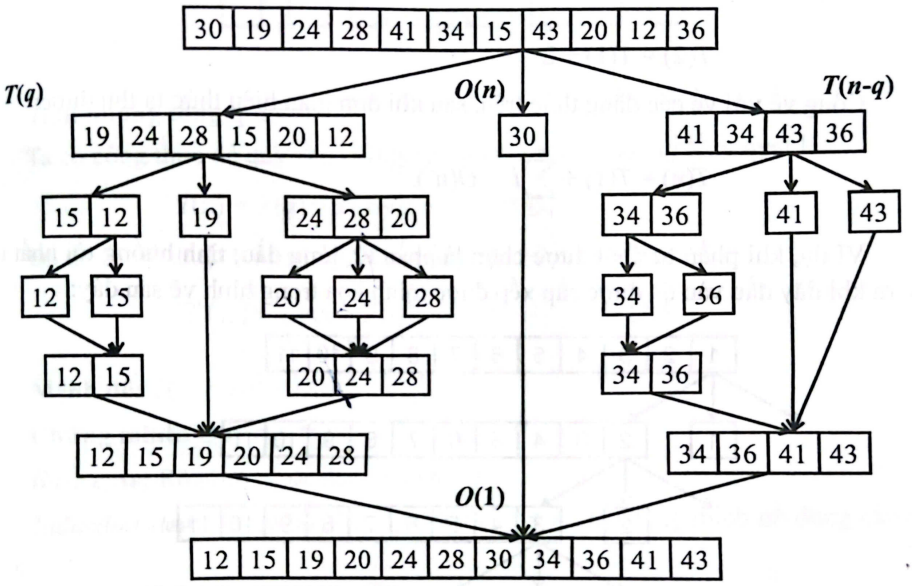
- Average case: độ phức tạp $\mathcal{O}(n \log k)$ với k là số runs của mảng.
- Hiệu hơn tốt hơn Merge sort khi dữ liệu đã gần sắp xếp, giảm số phép merge (chỉ merge các runs cần thiết).

10. Quick sort

10.1. Ý tưởng thuật toán

- Neo đệ quy: Nếu dãy chỉ còn không quá một phần tử thì là nó dãy đã được sắp xếp và trả lại ngay dãy này mà không phải làm gì cả.
- Chia: Chọn một phần tử trong dãy (ở đây ta chọn phần tử trung vị của a[low], a[mid], a[right]) và gọi nó là phần tử chốt *pivot*. Chia dãy đã cho ra thành hai dãy con: dãy con trái *low* gồm những phần tử không lớn hơn phần tử chốt, dãy con phải *high* gồm những phần tử không nhỏ hơn phần tử chốt. Thao tác này được gọi là Partition (phân đoạn).
- Trị: Lặp lại một cách đệ quy thuật toán đố với hai dãy con *low* và *high*.
- Tổng hợp: Dãy được sắp xếp là *low pivot high*.

10.2. Ví dụ thuật toán



Hình 1.10: Quick Sort

10.3. Code tham khảo

```
int medianOfThree(vector<int>& a, int low, int high) {
    int mid = low + (high - low) / 2;

    if (a[low] > a[mid]) swap(a[low], a[mid]);
    if (a[low] > a[high]) swap(a[low], a[high]);
    if (a[mid] > a[high]) swap(a[mid], a[high]);
    swap(a[mid], a[high]);
    return a[high];
}
int partition(vector<int>& a, int low, int high) {
    int pivot = medianOfThree(a, low, high);
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] < pivot) {
            i++;
```

```
            swap(a[i], a[j]);
        }
    }
    swap(a[i + 1], a[high]);
    return i + 1;
}
void quickSort(vector<int>& a, int low, int high) {
    if (low < high) {
        int pivot = partition(a, low, high);
        quickSort(a, low, pivot - 1);
        quickSort(a, pivot + 1, high);
    }
}
```

10.4. Phân tích thuật toán

- Best case: khi *pivot* chia mảng thành 2 phần gần đều nhau, độ phức tạp $\mathcal{O}(n \log n)$.
- Wost case: khi *pivot* chia không đều mảng (ví dụ mảng đã được sắp xếp hoặc có nhiều phần tử giống nhau, độ phức tạp $\mathcal{O}(n^2)$.
- Average case: mỗi lần *pivot* chia mảng thành 2 phần tương đối đều nhau, độ phức tạp $\mathcal{O}(n \log n)$.
- Việc chọn *pivot* bằng phương pháp Median of Three giúp giảm nguy cơ chọn *pivot* xấu (như một phần tử đầu tiên hoặc cuối cùng của một mảng đã sắp xếp hoặc gần sắp xếp), giúp tăng khả năng chia mảng gần đều hơn và cải thiện đáng kể hiệu suất của thuật toán.

11. std::sort

11.1. Nguyên lý hoạt động

std::sort là sự kết hợp của 3 thuật toán:

1. Quick sort: thuật toán chính, hiệu quả với dữ liệu ngẫu nhiên.

2. Heap sort: được sử dụng khi Quick sort rơi vào trường hợp tệ.

3. Insertion sort: dùng cho các dãy nhỏ (khoảng ≥ 16 phần tử).

11.2. Phân tích thuật toán

- Độ phức tạp luôn là $\mathcal{O}(n \log n)$ vì chuyển sang thuật toán tối ưu khi cần thiết.
- Làm việc với nhiều cấu trúc dữ liệu như vector, array, string,...
- Hiệu suất gần bằng các thuật toán được thiết kế riêng cho từng kiểu dữ liệu đầu vào.

12. Radix sort

12.1. Ý tưởng thuật toán

- Nhóm các khóa có cùng chữ số thứ i thành một nhóm đây là bước phân bố.
- Kết hợp các nhóm này lại thành một dãy duy nhất. Quá trình phân bố - kết hợp được thực hiện cho đến khi hết các chữ số.
- Có hai phương pháp chính: Least significant digit đi từ phải qua trái và Most significant digit đi từ trái qua phải. Ở đây nhóm chúng em chọn phương pháp đi từ phải qua trái (từ hàng đơn vị).

- Phân bố theo nhóm hàng đơn vị {170, 045, 075, 090, 802, 002, 024, 066}
  0: 170 090
  1:
  2: 002 802
  3:
  4: 024
  5: 045 075
  6: 066
  7:
  8:
  9:
- Gộp các nhóm {170, 090, 002, 802, 024, 045, 075, 066}

Hình 1.11: Step 1

- Phân bố theo nhóm hàng chục {170, 090, 002, 802, 024, 045, 075, 066}
  0: 002, 802
  1:
  2: 024
  3:
  4: 045
  5:
  6: 066
  7: 170, 075
  8:
  9: 090
- Gộp các nhóm {002, 802, 024, 045, 066, 170, 075, 090}

Hình 1.12: Step 2

- Phân bố theo nhóm hàng trăm {002, 802, 024, 045, 066, 170, 075, 090}
  0: 002, 024, 045, 066, 075, 090
  1: 170
  2:
  3:
  4:
  5:
  6:
  7:
  8: 802
  9:
- Gộp các nhóm {002, 024, 045, 066, 075, 090, 170, 802}

Hình 1.13: Step 3

12.3. Code tham khảo

```
int findMax(vector<int>& a) {
    int ans = INT_MIN;
    for (int num : a) {
        if (num > ans) {
            ans = num;
        }
    }
    return ans;
}
int findDigits(int num) {
    if (num == 0) return 1;
    int cnt = 0;
    while (num != 0) {
        num /= 10;
        cnt++;
    }
    return cnt;
```

```
}
void radixSort(vector<int>& a) {
    int  n = findDigits(findMax(a));
    for (int i = 0; i < n; i++) {
        vector<vector<int>>bin(10);
        for (int num : a) {
            int digit = (num / (int)pow(10, i)) % 10;
            bin[digit].push_back(num);
        }
        int idx = 0;
        for (int j = 0; j <= 9; j++) {
            for (int num : bin[j]) {
                a[idx++] = num;
            }
        }
    }
}
```

12.4. Phân tích thuật toán

- Độ phức tạp $\mathcal{O}(kn)$, k là số chữ số của số lớn nhất.
- Hiệu suất tốt với số nguyên có phạm vi hẹp và ổn định trong mọi trường hợp.
- Chỉ áp dụng cho số nguyên, nếu muốn làm việc với số thực hoặc chuỗi, cần phải chuyển đổi.
- Tốn bộ nhớ và kém hiệu quả khi số k lớn.

13. Counting sort

13.1. Ý tưởng thuật toán

- Đếm tần suất xuất hiện của từng giá trị trong mảng.
- Tái tạo mảng đã sắp xếp dựa trên tần suất.

13.2. Ví dụ thuật toán

```
Mảng ban đầu: a = [4, 2, 2, 8, 3, 3, 1]
→ Bước 1: min = 1, max = 8.
→ Bước 2: Khởi tạo: freq = [0, 0, 0, 0, 0, 0, 0, 0] (size = 8)
→ Bước 3: Đếm: freq = [1, 0, 2, 2, 1, 0, 0, 1]
→ Bước 4: Gán lại vào mảng: a = [1, 2, 2, 3, 3, 4, 8]
```

Hình 1.14: Counting sort

13.3. Code tham khảo

```cpp
int findMax(vector<int>& a) {
    int ans = INT_MIN;
    for (int num : a) {
        if (num > ans) {
            ans = num;
        }
    }
    return ans;
}
int findMin(vector<int>& a) {
    int ans = INT_MAX;
    for (int num : a) {
        if (ans > num) ans = num;
    }
    return ans;
}
void countingSort(vector<int>& a) {
    int min = findMin(a), max = findMax(a);
    vector<int>freq(max - min + 1, 0);
    for (int num : a) {
        freq[num - min]++;
    }
    int idx = 0;
```

```
    for (int i = 0; i < freq.size(); i++) {
        while (freq[i] > 0) {
            a[idx] = min + i;
            freq[i]--;
            idx++;
        }
    }
}
```

13.4. Phân tích thuật toán

- Độ phức tạp $\mathcal{O}(n + k)$, k là kích thước phạm vi giá trị (max - min + 1).
- Hoạt động cực kì nhanh và ổn định với phạm vi giá trị k nhỏ.
- Chỉ áp dụng cho số nguyên, nếu muốn làm việc với số thực hoặc chuỗi, cần phải chuyển đổi.
- Tốn bộ nhớ và kém hiệu quả khi số k lớn.

ĐÁNH GIÁ CÁC THỰC NGHIỆM

1. ĐÁNH GIÁ THỰC NGHIỆM 1

1.1. Dữ liệu đánh giá

Các mảng kiểu integer có độ lớn mảng là: $10^4, 2.10^4, 6.10^4, 8.10^4, 10.10^4, 12.10^4, 14.10^4, 16.10^4$ và 20.10^4

Tại mỗi mức độ lớn mảng, ta tạo ra các mảng ngẫu nhiên theo từng tiêu chí:

- Random Array
- Already Sorted Array
- Reverse Sorted Array
- Nearly Sorted Array

Code khởi tạo:

```
void makeRandomArray(vector<int>& vec, int n, int k) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<>dis(0, k);
    for (int& num : vec) {
        num = dis(gen);
    }
}
void makeAlreadySortedArray(vector<int>& vec, int n, int k) {
    makeRandomArray(vec, n, k);
    sort(vec.begin(), vec.end());
}
void makeReverseSortedArray(vector<int>& vec, int n, int k) {
    makeRandomArray(vec, n, k);
```

```
15      sort(vec.begin(), vec.end(), greater<int>());
16  }
17  void makeNearlySortedArray(vector<int>& vec, int n, int k) {
18      makeRandomArray(vec, n, k);
19      sort(vec.begin(), vec.end());
20      double sorted;
21      cout << "Input Ratio has been sorted: ";
22      cin >> sorted;
23
24      random_device rd;
25      mt19937 gen(rd());
26      uniform_int_distribution<>indexDis(0, n - 1);
27
28      int numSwaps = n * (1.0 - sorted);
29      for (int i = 1; i <= numSwaps/2; i++) {
30          swap(vec[indexDis(gen)], vec[indexDis(gen)]);
31      }
32  }
```

1.2. Cách đánh giá dữ liệu

Ở từng tiêu chí, ta cho sử dụng các thuật toán sắp xếp và lấy thời gian trung bình sắp xếp các mảng của mỗi thuật toán rồi so sánh chúng với nhau. Từ đó đưa ra các đánh giá về thuật toán phù hợp với các kiểu mảng.

1.3. Các thuật toán được dùng

Các thuật toán được sử dụng tại thực nghiệm 1:

- Shell Sort
- Heap Sort
- Merge Sort
- Natural Merge Sort

- Quick Sort
- std::sort
- Radix Sort
- Counting Sort
- Selection sort
- Insertion sort,
- Binary Insertion sort
- Bubble sort
- Shaker sort

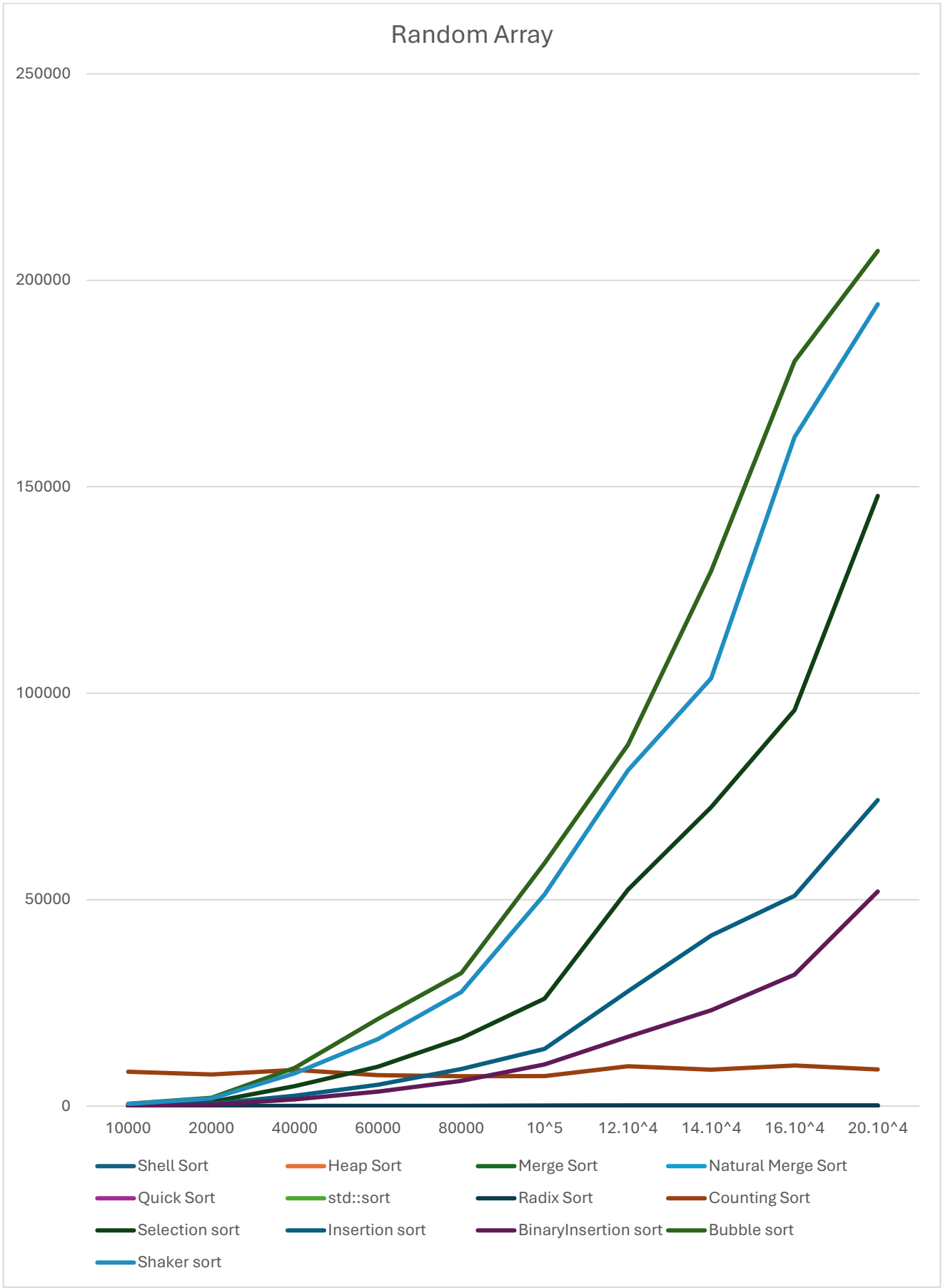
1.4. Cấu hình thực nghiệm

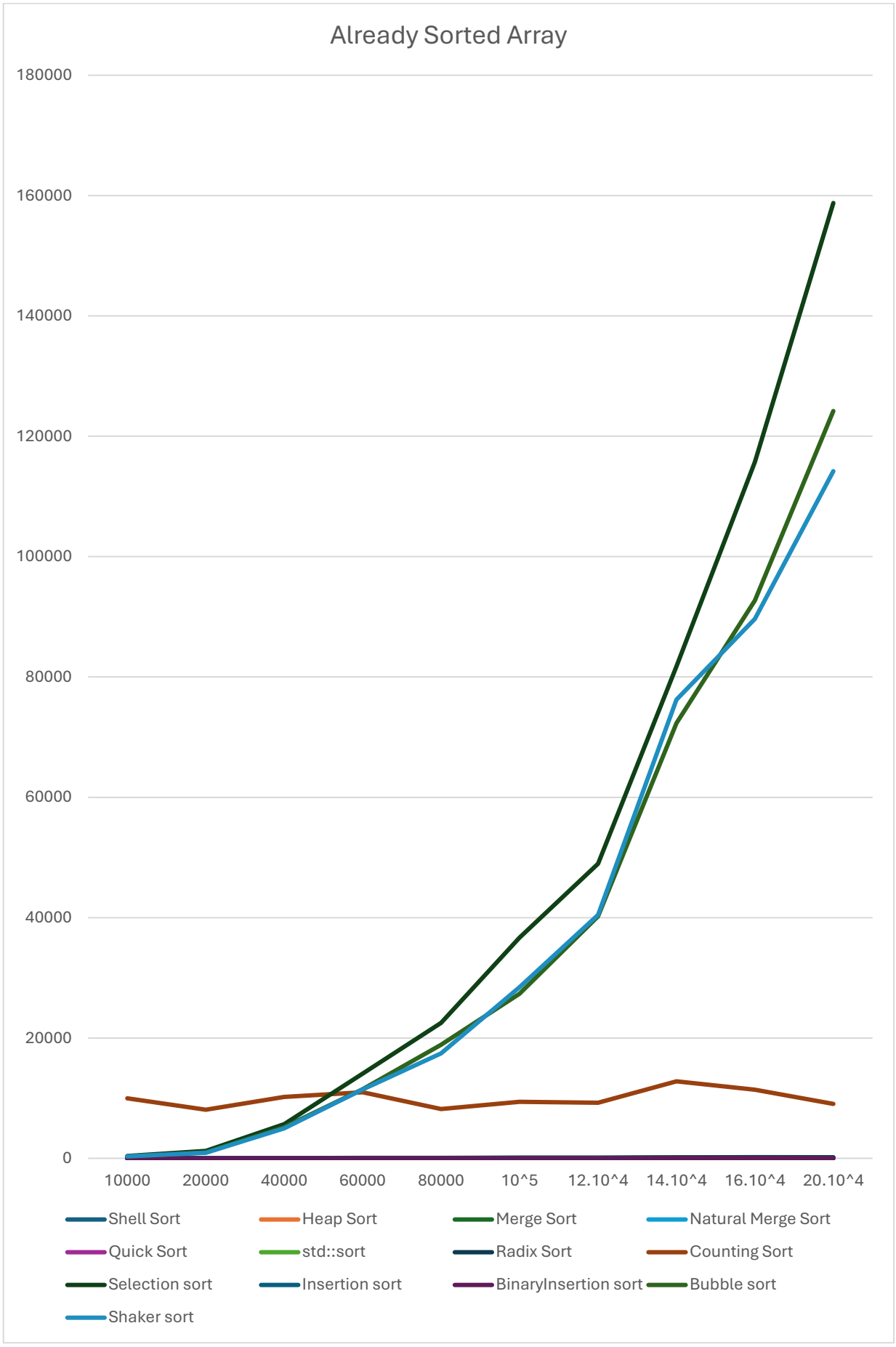
- CPU: Intel Core i7-1065G7
- RAM: 20 GB
- Clock Speed: 1.3-3.9 GHz

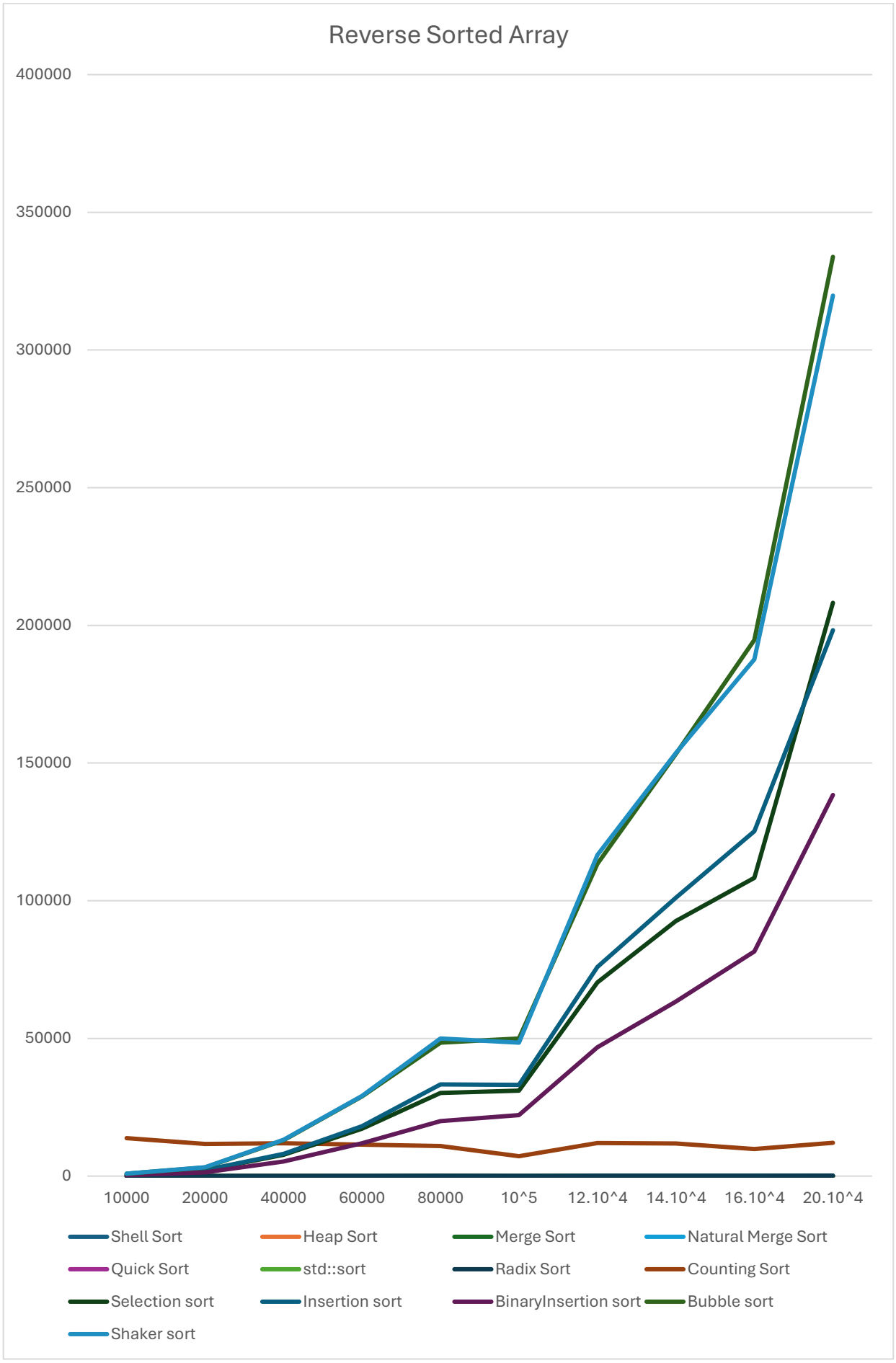
1.5. Kết quả thực nghiệm

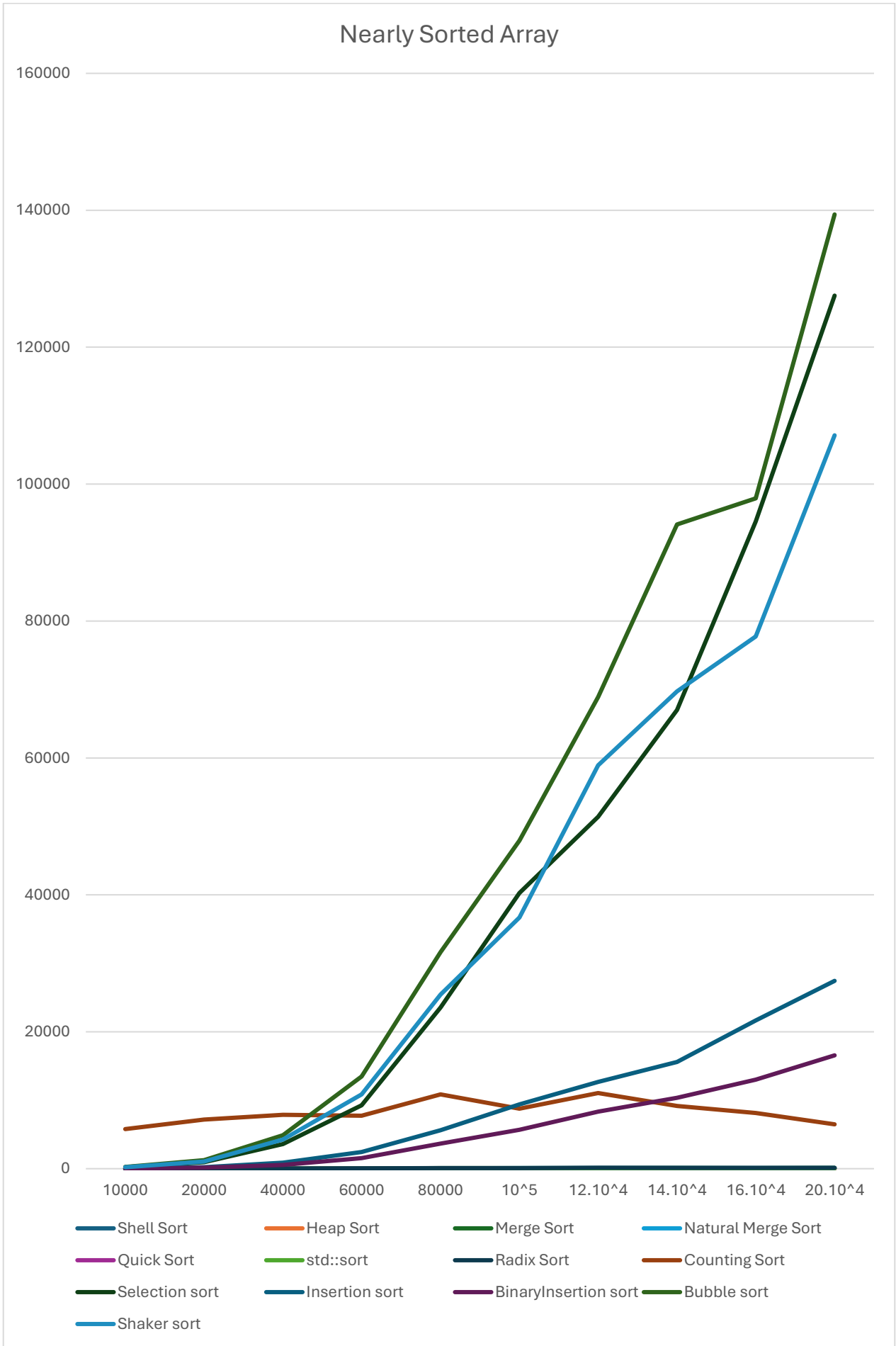
Dataset	Algorithm	10^4	2.10^4	4.10^4	6.10^4	8.10^4	10.10^4	12.10^4	14.10^4	16.10^4	20.10^4
	Shell Sort	7.60	7.80	19.60	26.60	39.20	43.40	70.60	75.80	114.10	120.10
	Heap Sort	0.40	3.20	7.20	8.00	14.00	20.00	34.20	38.60	36.00	46.20
	Merge Sort	12.20	12.40	20.40	35.00	38.20	60.00	112.60	115.40	145.60	144.00
	Natural Merge Sort	1.20	4.40	8.60	10.80	19.20	24.60	40.20	44.80	52.20	57.40
	Quick Sort	3.40	5.40	11.20	15.20	26.60	32.40	51.20	49.00	60.40	62.40
	std::sort	0.80	1.40	2.80	3.80	6.20	7.60	11.40	11.20	14.40	14.10
Random	Radix Sort	10.20	9.80	22.20	24.00	43.20	44.80	73.00	75.80	114.80	118.20
	Counting Sort	8321.20	7668.80	8766.00	7494.80	7233.00	7276.20	9636.60	8818.80	9632.00	8872.00
	Selection Sort	335.60	1022.60	4830.80	9532.40	16446.80	26044.20	52419.60	72399.60	95902.40	147787.20
	Insertion Sort	146.40	351.20	1026.20	2531.80	4036.40	6232.00	10763.20	14629.80	19383.00	30063.40
	Binary Insertion Sort	146.40	351.20	1026.20	2531.80	4036.40	6232.00	10763.20	14629.80	19383.00	30063.40
	Bubble Sort	556.40	2028.20	9286.20	21072.60	40246.40	58861.00	108748.40	152295.80	203018.80	313727.40
	Shaker Sort	519.00	1841.40	7929.80	16232.00	26736.00	40512.00	63223.20	83629.80	110339.20	169425.20
	Shell Sort	1.40	1.60	5.20	11.00	11.40	15.40	21.60	32.40	31.00	32.80
	Heap Sort	1.40	3.80	9.20	12.20	16.20	22.80	37.20	51.80	55.00	55.60
	Merge Sort	6.60	9.80	24.60	46.80	42.30	70.60	87.20	114.60	133.00	150.00
	Natural Merge Sort	0.00	0.00	0.00	0.00	0.00	0.00	0.40	1.20	0.60	1.40
	Quick Sort	1.40	1.20	4.40	6.60	6.60	14.20	15.20	26.40	23.40	21.80
	std::sort	5.00	1.80	6.40	10.40	10.00	16.80	21.80	37.80	38.40	35.40
Already	Radix Sort	4.80	13.20	32.80	62.40	52.40	90.60	103.80	164.80	162.80	170.40
Sorted	Counting Sort	959.20	3072.60	16168.80	30840.80	53149.60	90969.60	193135.80	171401.40	227925.00	352767.60
	Selection Sort	385.40	1223.40	5685.20	11076.40	19089.20	30368.60	60568.60	83147.40	110731.00	171587.60
	Insertion Sort	0.00	0.00	0.00	0.00	0.60	0.60	0.80	1.40	2.20	17.20
	Binary Insertion Sort	0.20	0.40	2.00	3.80	5.80	6.80	9.80	14.60	16.20	17.20
	Bubble Sort	307.80	974.80	5167.00	11476.60	18877.20	27353.60	40211.00	76228.00	406896.60	124194.20
	Shaker Sort	330.40	919.90	4994.80	11434.40	17426.60	28437.80	40450.00	76321.00	406945.60	124201.20
	Shell Sort	2.60	4.20	9.00	15.80	26.00	16.20	35.60	36.60	39.40	63.40
	Heap Sort	2.20	4.20	9.20	16.80	23.60	15.60	36.60	36.00	47.60	62.80
	Merge Sort	11.40	17.80	34.40	54.20	67.00	52.00	110.40	122.00	117.60	154.00
	Natural Merge Sort	15.20	25.40	45.60	77.40	96.40	72.40	159.00	180.20	184.60	205.20
	Quick Sort	2.60	4.80	9.40	16.40	24.00	20.40	41.00	45.00	43.40	59.60
	std::sort	4.80	8.20	16.20	27.20	38.20	30.80	62.40	71.40	75.20	99.80
Reverse	Radix Sort	15.60	22.40	38.60	69.00	76.20	62.80	121.80	135.80	144.20	181.60
Sorted	Counting Sort	13783.00	11714.20	19116.80	11442.00	20932.00	7528.20	12038.40	11883.80	19855.60	12105.20
	Selection Sort	563.80	2110.60	7706.60	17241.00	30211.40	47075.80	70296.20	102635.80	125272.00	192608.80
	Insertion Sort	539.40	2186.80	8034.00	18147.60	32116.00	60776.60	75986.00	110106.00	142421.80	291981.80
	Binary Insertion Sort	526.60	1368.40	5028.60	11695.20	20403.20	31339.40	48748.60	70341.60	91816.20	141528.80
	Bubble Sort	865.20	3169.40	13023.20	29991.20	49445.20	74139.00	114748.60	163280.80	213844.60	329413.40
	Shaker Sort	835.60	3174.40	13148.20	29912.80	49498.20	74199.60	114878.60	163380.60	213844.60	329413.40
	Shell Sort	2.40	5.80	15.20	21.00	45.00	56.80	73.40	82.40	80.60	93.80
	Heap Sort	0.80	2.40	6.80	7.40	25.80	22.20	25.80	37.40	32.60	34.80
	Merge Sort	2.80	10.20	21.40	35.80	77.00	81.80	99.80	107.60	96.20	107.00
	Natural Merge Sort	2.00	8.40	17.60	26.60	61.40	70.20	92.60	82.80	82.20	91.20
	Quick Sort	1.80	2.40	6.00	9.20	20.60	23.00	31.40	30.20	31.20	35.00
	std::sort	1.00	2.60	6.80	10.80	21.80	24.20	33.60	30.40	31.60	35.20
Nearly	Radix Sort	4.20	13.20	25.20	39.60	83.80	92.60	134.80	108.80	110.60	122.20
Sorted	Counting Sort	5785.60	7191.40	7864.40	7746.60	10857.40	8759.40	11049.80	9175.40	8149.80	6489.40
	Selection Sort	232.40	949.00	3592.40	9247.40	23549.20	40269.80	51417.20	66998.40	94539.80	127532.60
	Insertion Sort	49.20	223.60	880.60	2421.80	5602.60	9382.00	12661.20	15589.40	21667.40	27436.40
	Binary Insertion Sort	31.20	142.60	555.80	1527.60	3659.00	5697.80	8345.80	10367.80	13004.60	16557.40
	Bubble Sort	256.80	1261.20	4886.40	13475.00	31639.20	47930.00	68894.60	94091.60	97906.20	139359.80
	Shaker Sort	217.00	1041.00	4264.00	10867.60	25436.40	36680.00	58919.80	69709.20	77756.00	107126.40

Bång 2.1: Exercise 1


Random Array
250000
200000
150000
100000
50000
0
10000
20000
40000
60000
80000
10^5
12.10^4
14.10^4
16.10^4
20.10^4
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort
Selection sort
Insertion sort
BinaryInsertion sort
Bubble sort
Shaker sort


Already Sorted Array
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
10000
20000
40000
60000
80000
10^5
12.10^4
14.10^4
16.10^4
20.10^4
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort
Selection sort
Insertion sort
BinaryInsertion sort
Bubble sort
Shaker sort


Reverse Sorted Array
400000
350000
300000
250000
200000
150000
100000
50000
0
10000
20000
40000
60000
80000
10^5
12.10^4
14.10^4
16.10^4
20.10^4
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort
Selection sort
Insertion sort
BinaryInsertion sort
Bubble sort
Shaker sort


Nearly Sorted Array
160000
140000
120000
100000
80000
60000
40000
20000
0
10000
20000
40000
60000
80000
10^5
12.10^4
14.10^4
16.10^4
20.10^4
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort
Selection sort
Insertion sort
BinaryInsertion sort
Bubble sort
Shaker sort

1.6. Nhận xét

- Nhận xét tổng quát:
  - Nhìn chung thời gian chạy các thuật toán tăng dần khi độ lớn mảng cần sắp xếp càng lớn.
  - Tại độ lớn mảng bằng 10000 và 20000 thời gian chạy khá nhanh nên không có sự khác biệt.
  - Counting Sort có thời gian chạy ổn định gần như không đổi mặc dù độ lớn mảng có tăng dần lên.
- Nhận xét riêng:
  - Random Array:
    - Khi độ lớn mảng lớn (trên 80000), Bubble Sort có thời gian chạy lâu nhất, trong khi đó Counting sort lại khá nhỏ và ổn định.
    - Khi độ lớn mảng vừa (khoảng 40000 – 80000), Binary Insertion Sort lại cho thời gian chạy nhanh nhất.
    - Độ lớn mảng nhỏ (dưới 40000), các thuật toán chạy với thời gian không chênh lệch nhau quá nhiều, Counting Sort cho thời gian chạy lâu hơn so với các thuật toán còn lại.
  - Already Sorted Array:
    - Khi độ lớn mảng lớn và vừa (trên 60000), các thuật toán chênh lệch nhau khá ít, Selection Sort cho thời gian chạy lâu nhất so với các thuật toán còn lại, Counting Sort cho thời gian chạy hiệu quả nhất.
    - Khi độ lớn mảng nhỏ (dưới 60000), Counting Sort cho thời gian chạy lâu nhất.
  - Reverse Sorted Array:
    - Các mảng chạy với thời gian gần tương tự nhau tại độ lớn mảng khoảng 50000.
    - Độ lớn mảng lớn (trên 60000), Bubble Sort cho thời gian chạy lâu nhất, Counting Sort cho thời gian chạy hiệu quả nhất.
    - Độ lớn mảng nhỏ (dưới 40000), các thuật toán đều chạy khá nhanh, Counting Sort cho thời gian chạy kém hiệu quả nhất.

– Nearly Sorted Array:

- Khi độ lớn mảng lớn (trên 60000), Bubble Sort cho thời gian chạy lâu nhất.
- Độ lớn mảng khoảng từ 40000 – 100000, Binary Insertion Sort cho thời gian chạy khá hiệu quả.

2. ĐÁNH GIÁ THỰC NGHIỆM 2

2.1. Dữ liệu đánh giá

Các mảng kiểu integer có độ lớn mảng là: $10^6, 2.10^6, 6.10^6, 8.10^6, 10.10^6, 12.10^6, 14.10^6, 16.10^6$ và 20.10^6

Tại mỗi mức độ lớn mảng, ta tạo ra các mảng ngẫu nhiên theo từng tiêu chí:

- Random Array
- Already Sorted Array
- Reverse Sorted Array
- Nearly Sorted Array

Code khởi tạo:

```
void makeRandomArray(vector<int>& vec, int n, int k) {
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<>dis(0, k);
    for (int& num : vec) {
        num = dis(gen);
    }
}
void makeAlreadySortedArray(vector<int>& vec, int n, int k) {
    makeRandomArray(vec, n, k);
    sort(vec.begin(), vec.end());
}
void makeReverseSortedArray(vector<int>& vec, int n, int k) {
```

```
    makeRandomArray(vec, n, k);
    sort(vec.begin(), vec.end(), greater<int>());
}
void makeNearlySortedArray(vector<int>& vec, int n, int k) {
    makeRandomArray(vec, n, k);
    sort(vec.begin(), vec.end());
    double sorted;
    cout << "Input Ratio has been sorted: ";
    cin >> sorted;

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<>indexDis(0, n - 1);

    int numSwaps = n * (1.0 - sorted);
    for (int i = 1; i <= numSwaps/2 ; i++) {
        swap(vec[indexDis(gen)], vec[indexDis(gen)]);
    }
}
```

2.2. Cách đánh giá dữ liệu

Ở từng tiêu chí, ta cho sử dụng các thuật toán sắp xếp và lấy thời gian trung bình sắp xếp các mảng của mỗi thuật toán rồi so sánh chúng với nhau. Từ đó đưa ra các đánh giá về thuật toán phù hợp với các kiểu mảng.

2.3. Các thuật toán được dùng

Các thuật toán được sử dụng tại thực nghiệm 2:

- Shell Sort
- Heap Sort
- Merge Sort

- Natural Merge Sort
- Quick Sort
- std::sort
- Radix Sort
- Counting Sort

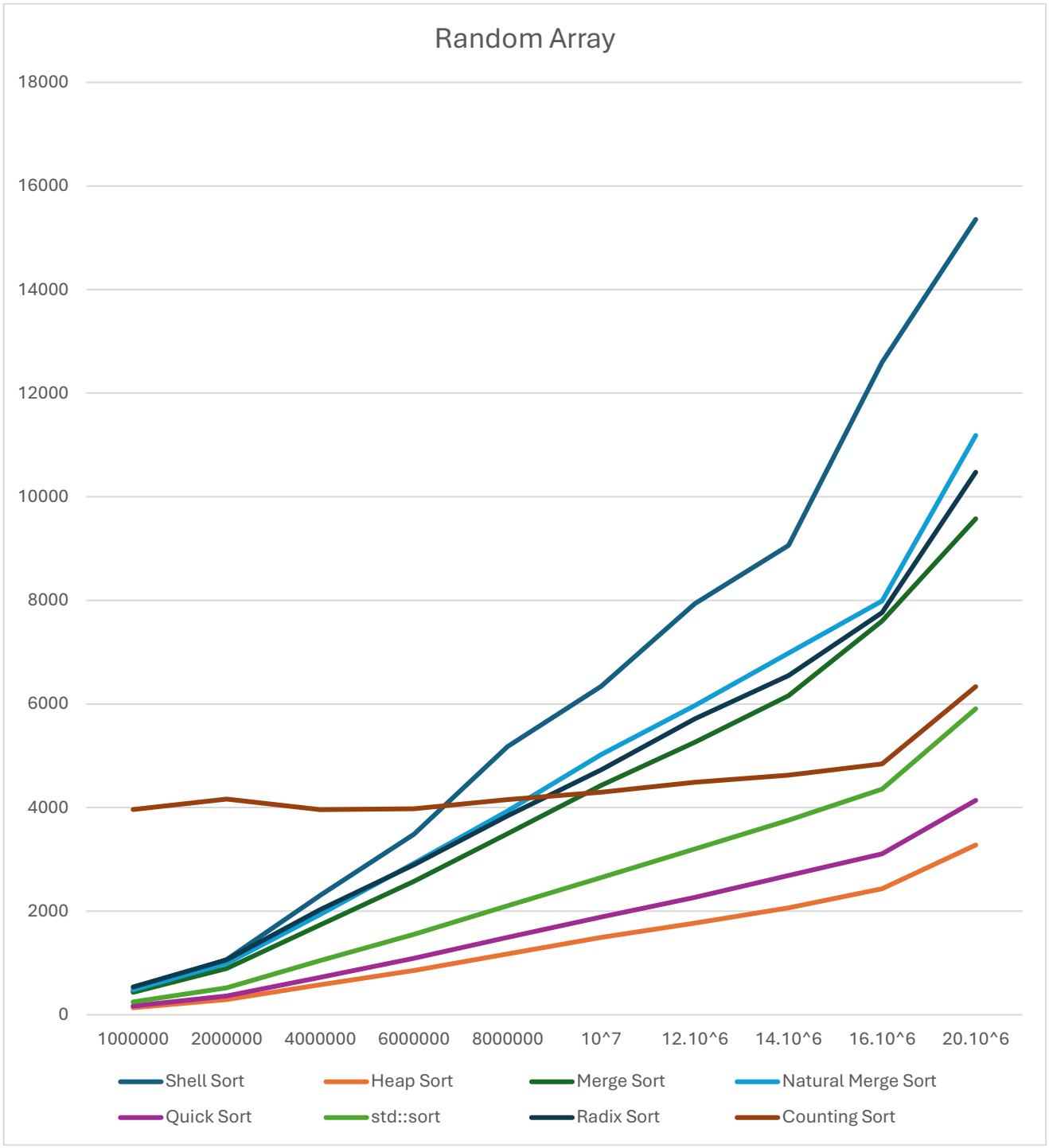
2.4. Cấu hình thực nghiệm

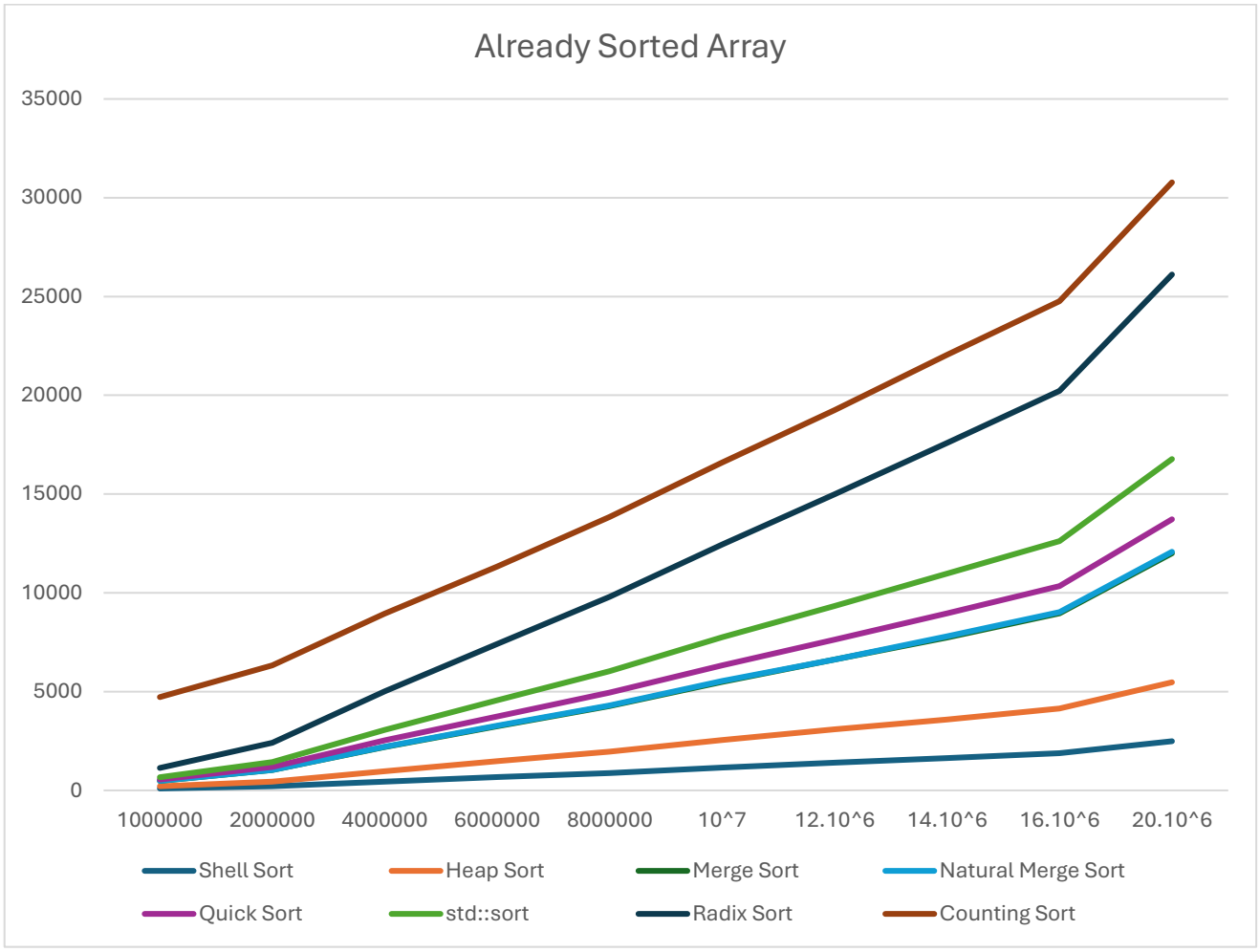
- CPU: AMD Ryzen 7 8845H w
- RAM: 32 GB
- Clock Speed: 3.3-4.9 GHz

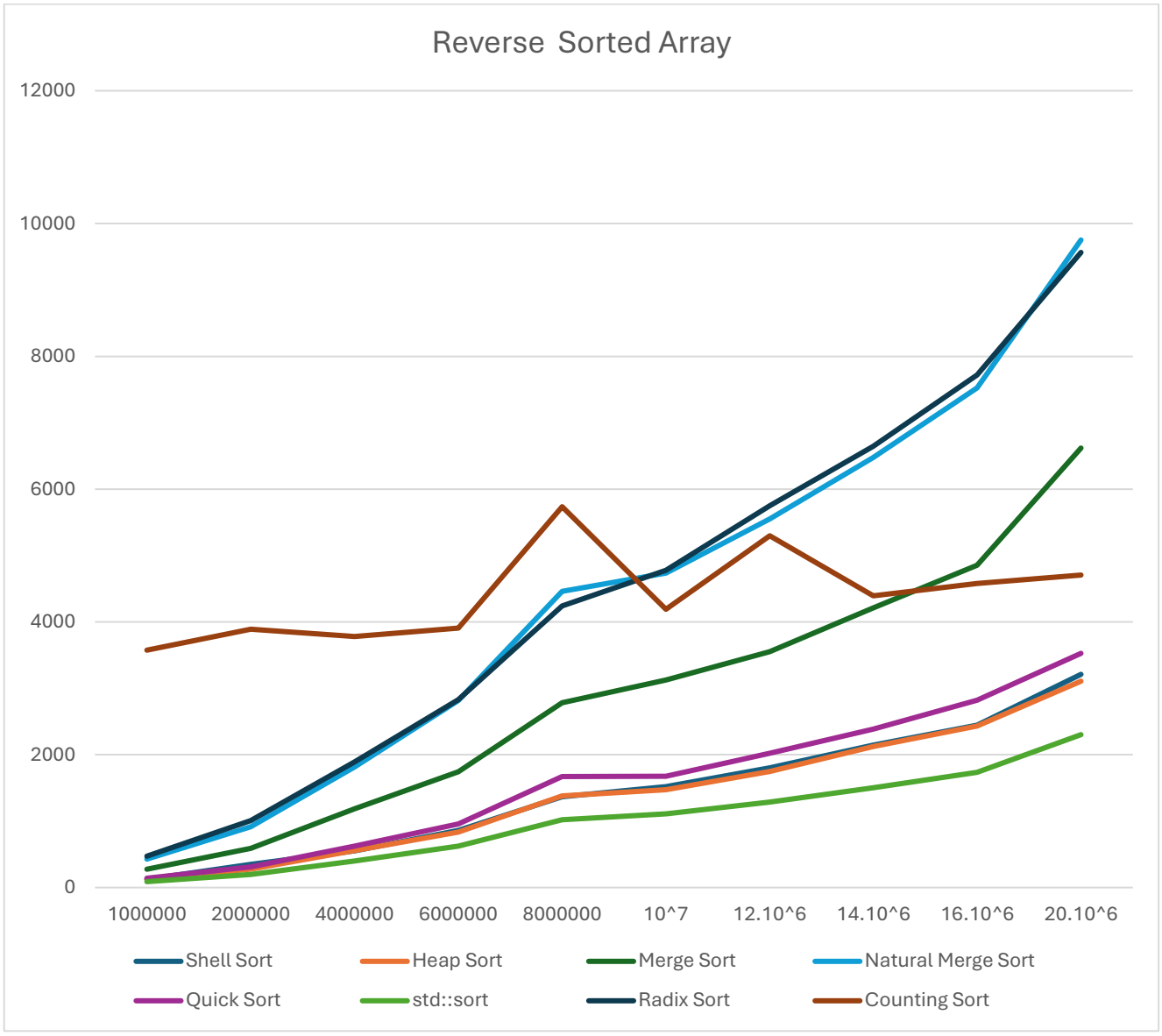
2.5. Kết quả thực nghiệm

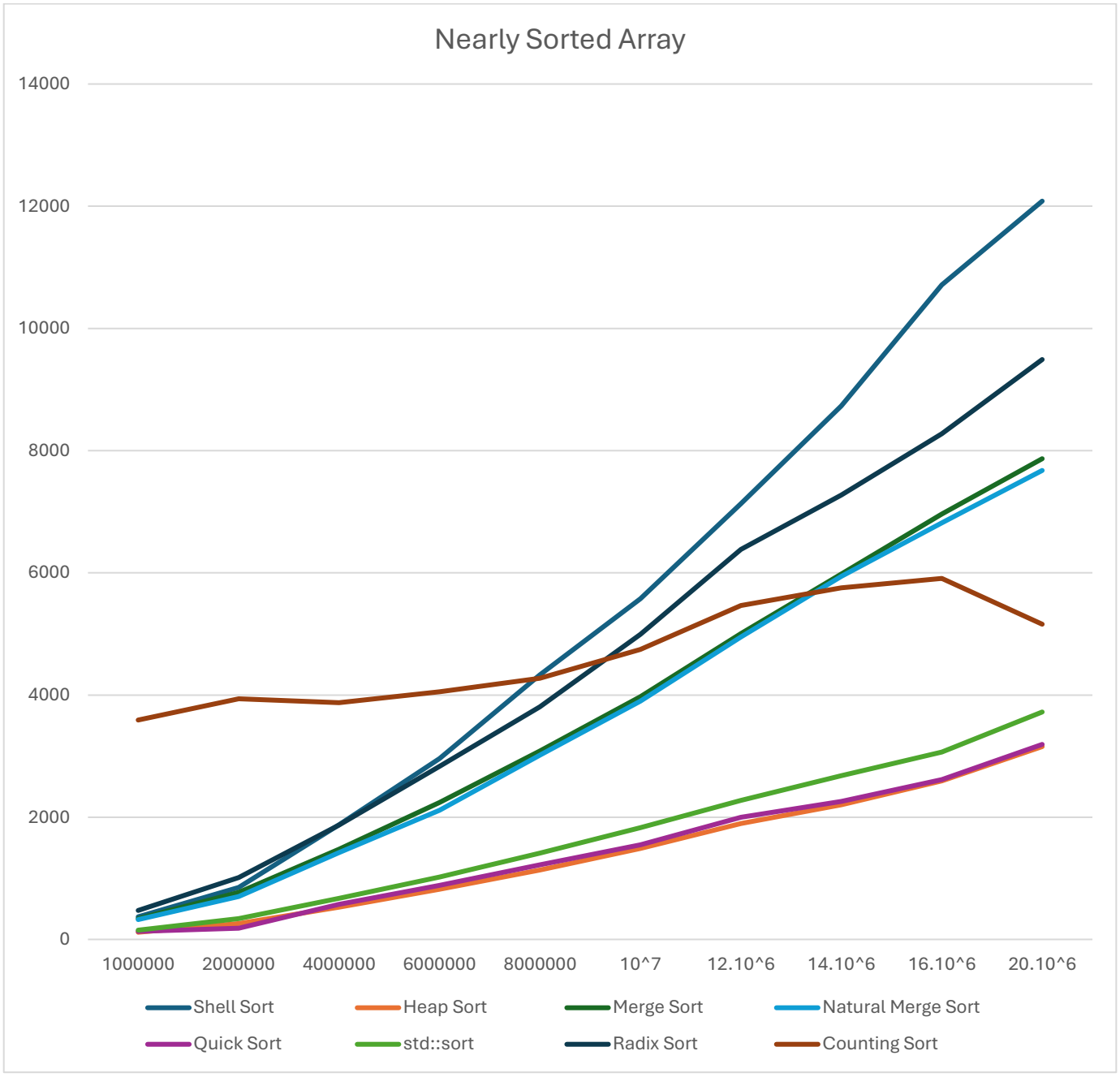
Dataset	Algorithm	10^6	2.10^6	4.10^6	6.10^6	8.10^6	10.10^6	12.10^6	14.10^6	16.10^6	20.10^6
Random	Shell Sort	458.7	1067.7	2303.8	3484.7	5177.4	6341.4	7935.8	9059.3	12599.7	15354.2
	Heap Sort	135.9	295.5	579	854.6	1173.7	1492.1	1768.7	2062.1	2435	3277.3
	Merge Sort	433.5	892.5	1733.1	2574.2	3495.4	4431.9	5260.5	6161.7	7399.4	9575.6
	Natural Merge Sort	480.2	991.3	1935.1	2922.5	3934.9	5022.1	5968	6979.9	7991.3	11184.1
	Quick Sort	169.6	364.2	724	1091	1492.7	1885.9	2265.1	2689.1	3107.3	4139.3
	std::sort	248.2	521.3	1048.7	1553.4	2106.1	2649.9	3198.6	3753.8	4335.2	5906.9
	Radix Sort	536.2	1062.1	2028.5	2893.4	3841.4	4728.1	5713.2	6550.9	7764	10474.3
	Counting Sort	3961.7	4165.5	3959.7	3975.1	4155.4	4296.1	4487.8	4625.6	4843.4	6335
Already sorted	Shell Sort	96.6	203.4	442.9	674.8	883.4	1148.2	1408.4	1631.2	1889.8	2488.8
	Heap Sort	113.3	242.9	531.6	811.4	1072.2	1396.2	1683.8	1957.4	2251.6	2982.8
	Merge Sort	269.9	581.4	1220.5	1765.8	2321	2950.2	3536.6	4157.4	4822	6530.2
	Natural Merge Sort	3.2	8.2	15.5	22.2	31	38.4	45.4	54.2	62	77.2
	Quick Sort	67.6	148.4	317.5	475.2	637.2	796	997.4	1169	1320.5	1643.6
	std::sort	116.1	248.5	538.2	816.6	1084.4	1431.4	1715.2	2001.4	2268.8	3050
	Radix Sort	469.1	981.7	1947.6	2843.4	3776.8	4692.2	5643.6	6620.4	7603.2	9340.2
	Counting Sort	3587	3910.1	3935.2	3926.8	4043	4139	4256	4464.1	4541.6	4665
Reverse sorted	Shell Sort	123.8	347.6	551.8	860.8	1368.8	1518.6	1803.67	2145.5	2443.5	3216.75
	Heap Sort	124	276.6	554.8	834	1382	1472	1747.67	2125.25	2433.25	3107.5
	Merge Sort	275.8	588.4	1186.8	1741.8	2784	2994.4	3553.33	4212	4854.5	6618.75
	Natural Merge Sort	428.2	913.4	1816.2	2816.8	4461.6	4734.6	5551	6477.25	7525	9726
	Quick Sort	140.4	312.8	624.2	958.2	1670.4	1677	2021.33	2387.75	2823	3527
	std::sort	88.8	196.8	401.6	624.4	1022	1107.8	1288.333	1502	1735.25	2302.5
	Radix Sort	472.2	1009	1893.6	2830.2	4243	4780	5751.33	6644	7718.75	9565
	Counting Sort	3575	3892	3779.8	3905.6	5735.6	4192.4	5297.67	4393.5	4579.5	4705.25
Nearly sorted	Shell Sort	370	851	1874.67	2961.33	4327.67	5575	7133.67	8731	10714.67	12083.67
	Heap Sort	117.4	261.67	527.33	822	1136.33	1489.33	1897.33	2205	2593.67	3156
	Merge Sort	340	770.67	1475.33	2241	3081.67	3971.33	5005.67	5979	6960.33	7867.67
	Natural Merge Sort	324.6	703.67	1422.33	2113	3017.67	3899.33	4949.67	5942	6916.33	7675.67
	Quick Sort	131.4	183	576.33	888.33	1221.33	1547.33	1999.33	2257	2616.33	3191.67
	std::sort	151.8	341	670.33	1022	1412.67	1831	2277.33	2677	3067	3722.33
	Radix Sort	474.2	1013.67	1872.33	2831.67	3807.33	4992.67	6381.67	7272.67	8276	9491
	Counting Sort	3590.8	3939.67	3875	4055	4273	4747.67	5467	5752.33	5908.67	5159.67

Bång 2.2: Exercise 2


Random Array
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1000000
2000000
4000000
6000000
8000000
10^7
12.10^6
14.10^6
16.10^6
20.10^6
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort


Already Sorted Array
35000
30000
25000
20000
15000
10000
5000
0
1000000
2000000
4000000
6000000
8000000
10^7
12.10^6
14.10^6
16.10^6
20.10^6
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort


Reverse Sorted Array
12000
10000
8000
6000
4000
2000
0
1000000
2000000
4000000
6000000
8000000
10^7
12.10^6
14.10^6
16.10^6
20.10^6
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort


Nearly Sorted Array
14000
12000
10000
8000
6000
4000
2000
0
1000000
2000000
4000000
6000000
8000000
10^7
12.10^6
14.10^6
16.10^6
20.10^6
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort
Radix Sort
Counting Sort

2.6. Nhận xét

- Nhận xét tổng quát:
  - Các thuật toán có thời gian chạy tăng dần khi độ lớn mảng tăng.
  - Khi độ lớn mảng nhỏ, Counting Sort luôn cho thời gian chạy lâu nhất.
  - Counting Sort có độ biến thiên thời gian chạy khi độ lớn mảng tăng là khá nhỏ so với các thuật toán khác.
- Nhận xét riêng:
  - Random Array:
    - Counting Sort có tốc độ thay đổi thời gian khá nhỏ khi độ lớn mảng tăng.
    - Khi độ lớn mảng tăng, Shell Sort cho tốc độ chạy chậm nhất, trong khi Heap Sort chạy khá nhanh.
  - Already Sorted Array:
    - Counting Sort luôn cho thời gian chạy lâu nhất trong khi Shell Sort cho thời gian chạy khá nhỏ ở mọi độ lớn của mảng.
  - Reverse Sorted Array:
    - Tại độ lớn mảng bằng 8000000, 10000000, và 12000000 đồ thị của Counting Sort có bị "biến dạng" một chút so với các đồ thị khác, điều này có thể là do quá trình tạo mảng ngẫu nhiên của mỗi máy tính khác nhau.
    - Khi mảng có độ lớn cao, Radix Sort và Natural Merge Sort cho thời gian chạy lâu nhất trong khi đó std::sort chạy khá nhanh.
  - Nearly Sorted Array:
    - Tại độ lớn của mảng lớn hơn 8000000, Shell Sort luôn cho thời gian chạy lâu nhất.
    - Tại mọi độ lớn của mảng, Quick Sort luôn chạy hiệu quả nhất.

3. ĐÁNH GIÁ THỰC NGHIỆM 3

3.1. Dữ liệu đánh giá

Sử dụng thông tin từ tài liệu `Oxford English Dictionary.txt`, khởi tạo được mảng các kí tự string.

Từ đó, ta tạo ra các mảng ngẫu nhiên theo từng tiêu chí:

- Random Array
- Already Sorted Array
- Reverse Sorted Array
- Nearly Sorted Array

Code khởi tạo:

```
void makeRandomArray(const string path, vector<string>& vec) {
    makeAlreadySortedArray(path, vec);

    int n = vec.size();
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<>indexDis(0, n - 1);

    for (int i = 1; i <= n/2; i++) {
        int idx1 = indexDis(gen), idx2;
        do {
            idx2 = indexDis(gen);
        } while (idx1 == idx2);
        swap(vec[idx1], vec[idx2]);
    }
}

void makeAlreadySortedArray(const string path, vector<string>& vec) {
    ifstream ifs(path);
```

```
20      if (!ifs.is_open()) {
21          cout << "Can't open file" << endl;
22          return;
23      }
24      string temp;
25      while (getline(ifs, temp)) {
26          if (temp.empty() || isSpaceOnly(temp)) {
27              continue;
28          }
29          vec.push_back(temp);
30      }
31      ifs.close();
32  }
33
34  void makeReverseSortedArray(const string path, vector<string>& vec) {
35      makeAlreadySortedArray(path, vec);
36      reverse(vec.begin(), vec.end());
37  }
38
39  void makeNearlySortedArray(const string path, vector<string>& vec) {
40      makeAlreadySortedArray(path, vec);
41
42      double sorted;
43      cout << "Input Ratio has been sorted: ";
44      cin >> sorted;
45
46      int n = vec.size();
47      random_device rd;
48      mt19937 gen(rd());
49      uniform_int_distribution<>indexDis(0, n - 1);
50
51      int numSwaps = n * (1.0 - sorted) / 2;
52      for (int i = 1; i <= numSwaps; i++) {
```

```
53          int idx1 = indexDis(gen), idx2;
54          do {
55              idx2 = indexDis(gen);
56          } while (idx1 == idx2);
57          swap(vec[idx1], vec[idx2]);
58      }
59  }
```

3.2. Cách đánh giá dữ liệu

Ở từng tiêu chí, ta cho sử dụng các thuật toán sắp xếp và lấy thời gian trung bình sắp xếp các mảng của mỗi thuật toán rồi so sánh chúng với nhau. Từ đó đưa ra các đánh giá về thuật toán phù hợp với các kiểu mảng.

3.3. Các thuật toán được dùng

Các thuật toán được sử dụng tại thực nghiệm 3:

- Shell Sort
- Heap Sort
- Merge Sort
- Natural Merge Sort
- Quick Sort
- std::sort
- Radix Sort
- Counting Sort
- Selection sort
- Insertion sort,
- Binary Insertion sort

- Bubble sort
- Shaker sort

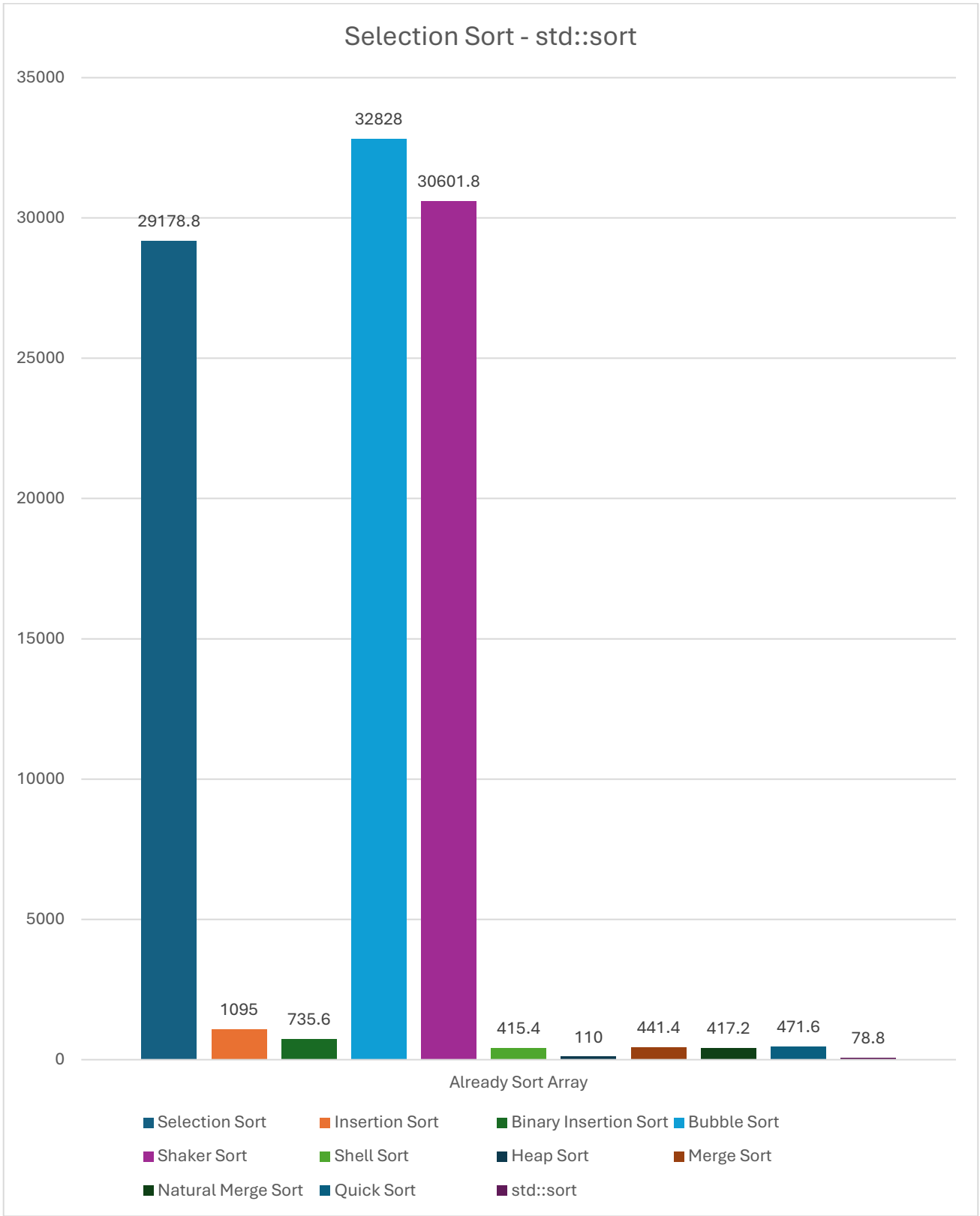
3.4. Cấu hình thực nghiệm

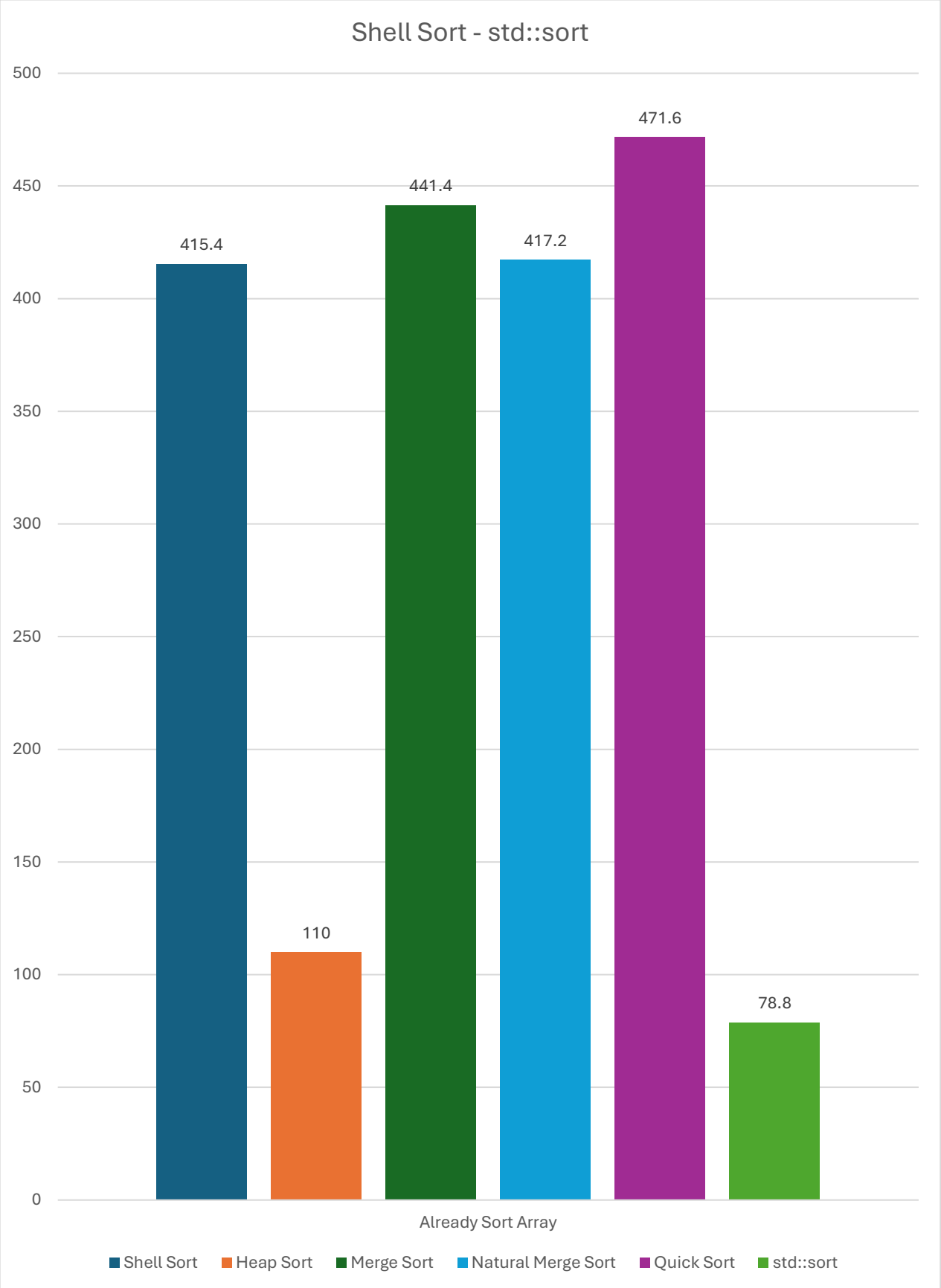
- CPU: AMD Ryzen 7 7840HS
- RAM: 32 GB
- Clock Speed: 3.8-5.1 GHz

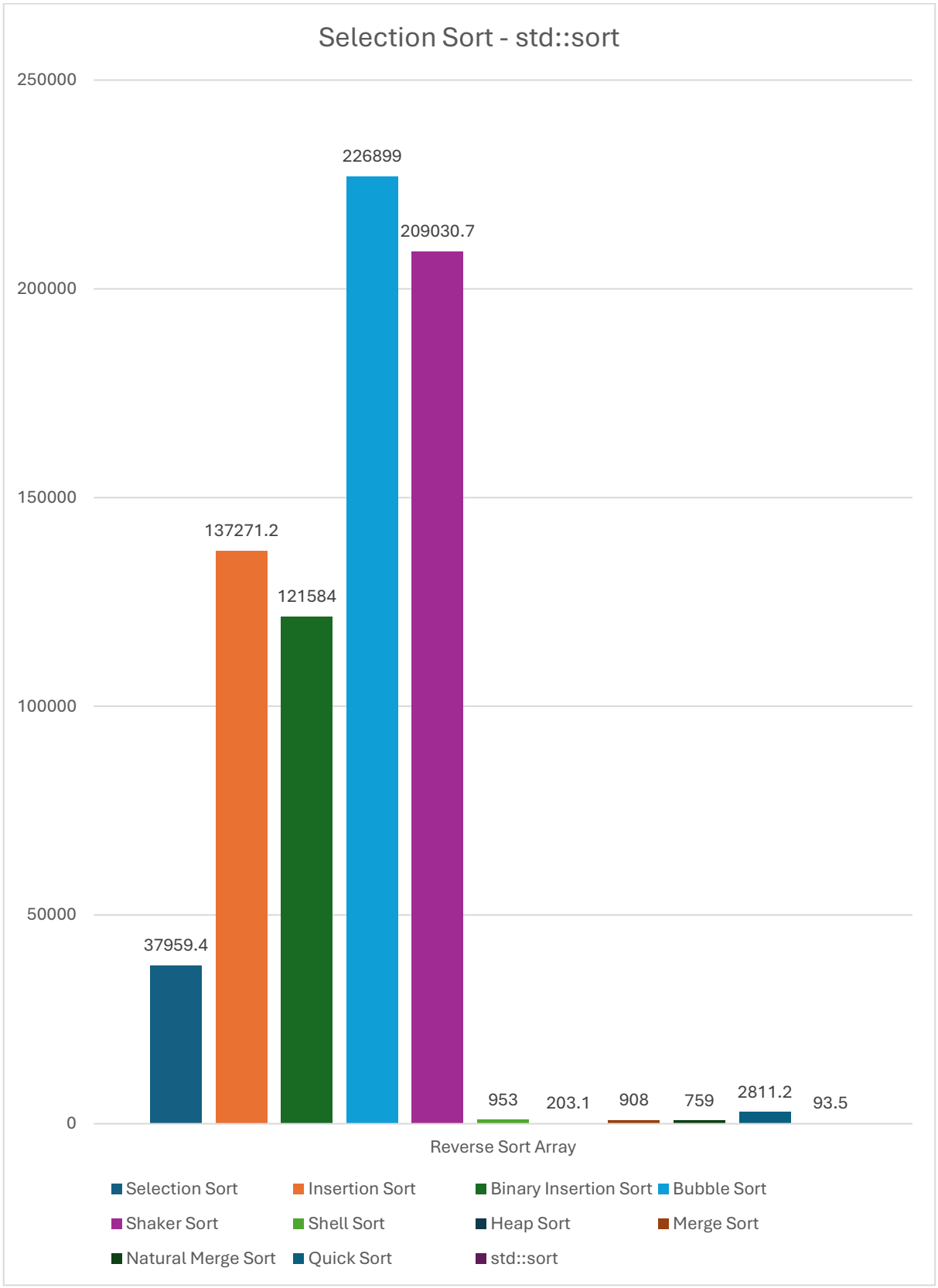
3.5. Kết quả thực nghiệm

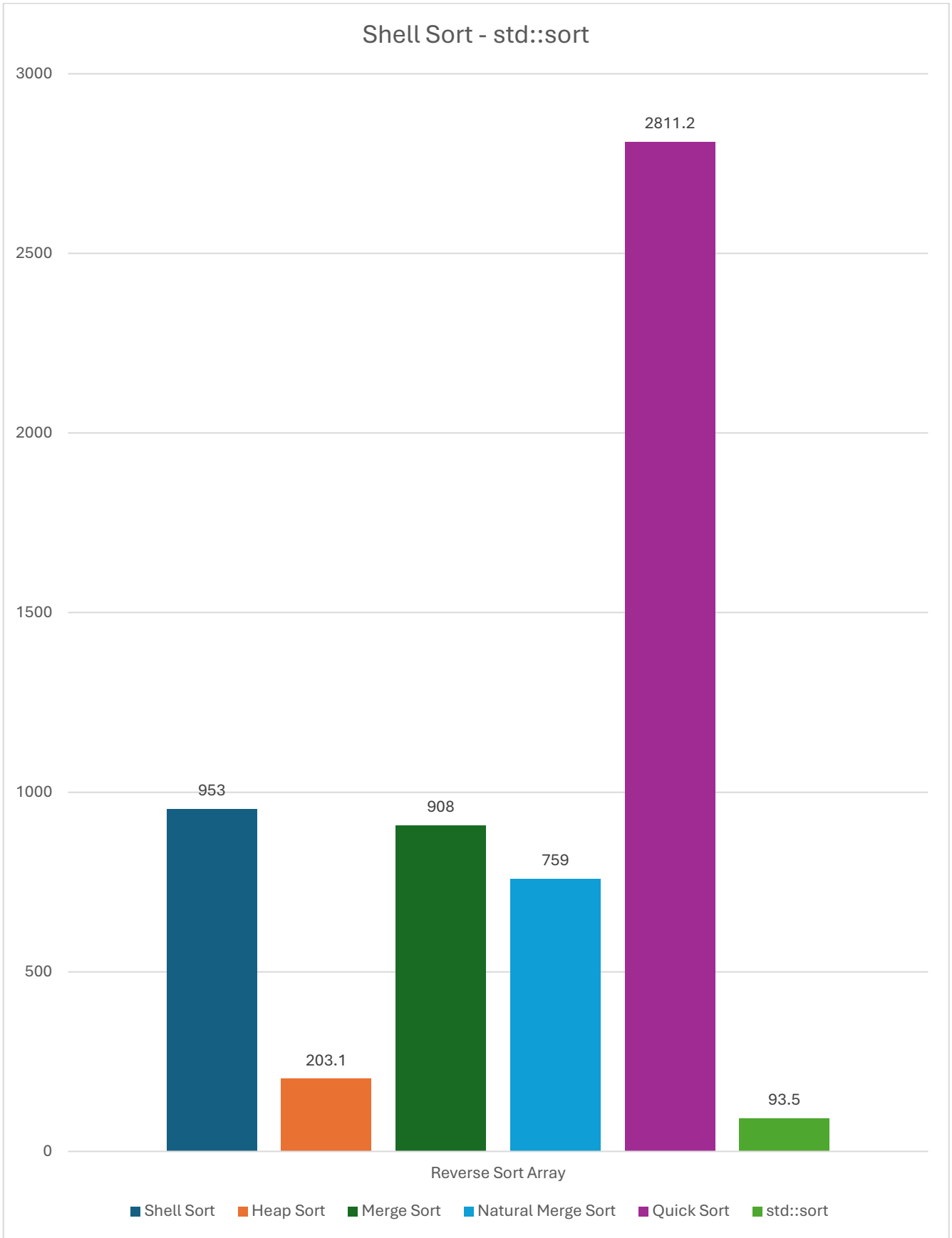
Algorithm	Already Sort Array	Reverse Sort Array	Nearly Sort Array	Random Array
Selection Sort	29178.8	37959.4	58678.2	29914.6
Insertion Sort	1095	137271.2	18185	31221.7
Binary Insertion Sort	735.6	121584	11588.2	20855.5
Bubble Sort	32828	226899	112555.2	71899
Shaker Sort	30601.8	209303.7	91896.8	61813.2
Shell Sort	415.4	953	771	505.33
Heap Sort	110	203.1	188.6	114.4
Merge Sort	441.4	908	811	458.4
Natural Merge Sort	417.2	759	825.6	521.5
Quick Sort	471.6	2811.2	167	90.33
std::sort	78.8	93.5	176.6	108.8
Radix Sort	580673.4	560334.7	889945.2	561005
Counting Sort	451691.4	457892.8	810557.2	461237.2

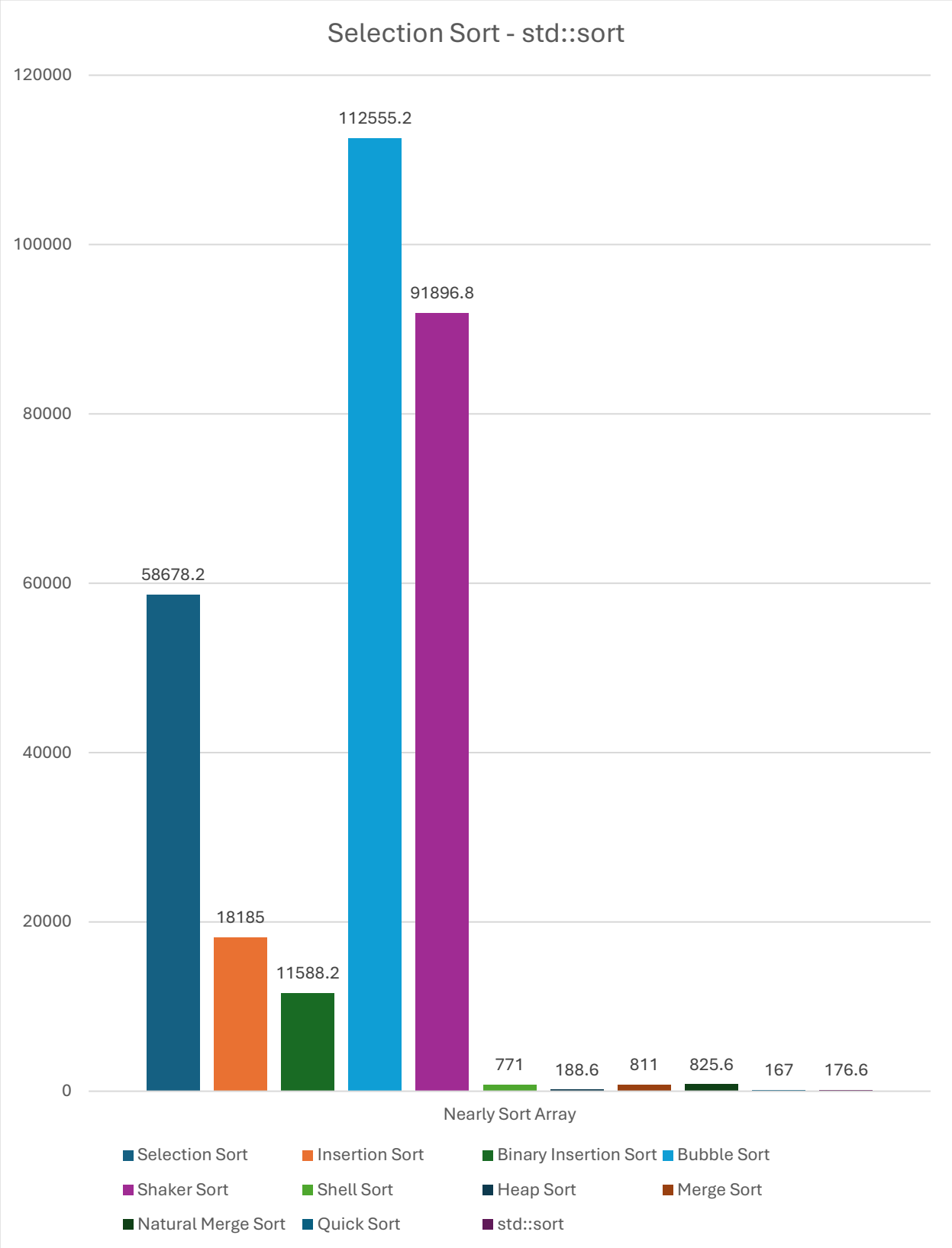
Bảng 2.3: Exercise 3

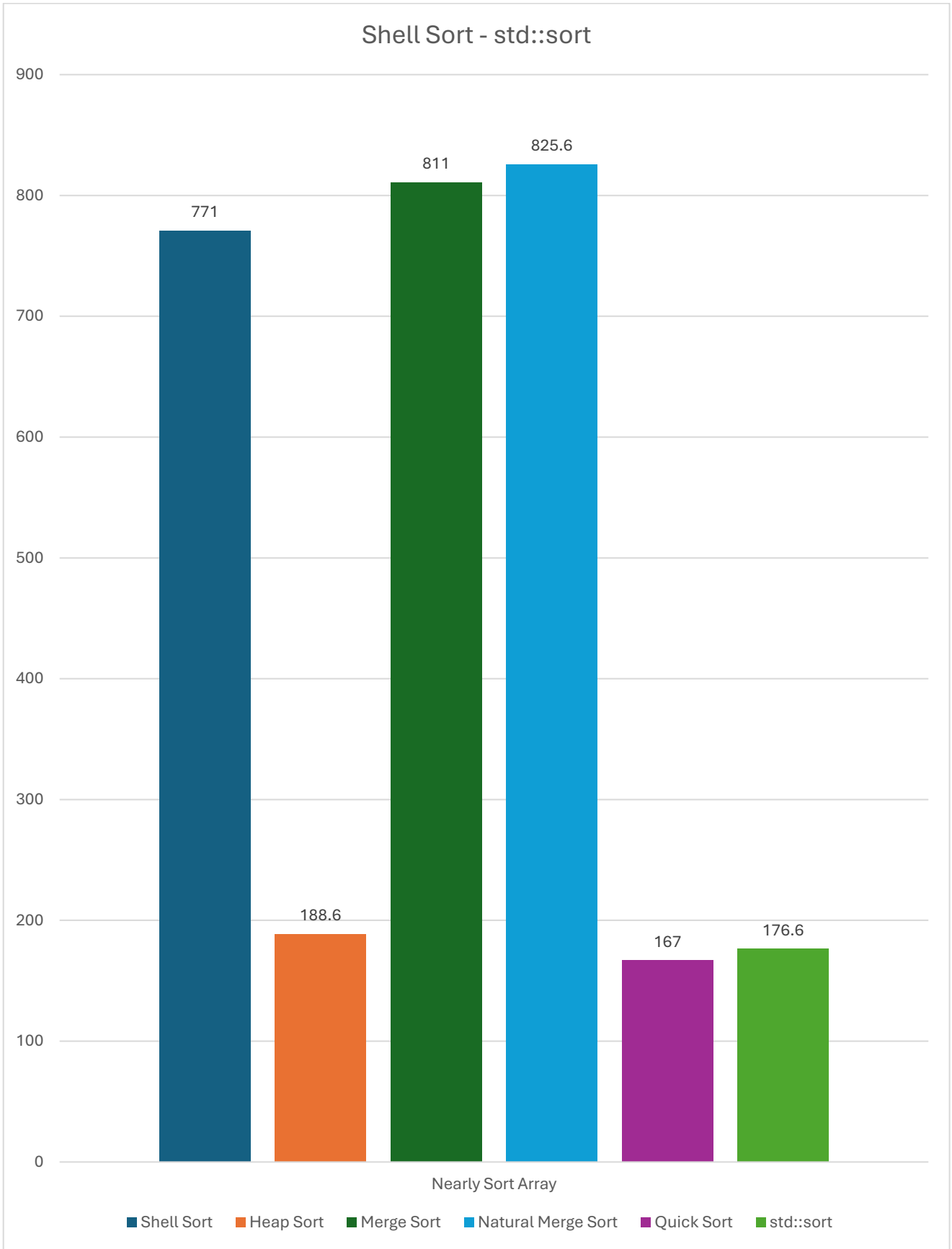

Selection Sort - std::sort
35000
30000
25000
20000
15000
10000
5000
0
29178.8
1095
735.6
32828
30601.8
415.4
110
441.4
417.2
471.6
78.8
Already Sort Array
Selection Sort
Insertion Sort
Binary Insertion Sort
Bubble Sort
Shaker Sort
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort

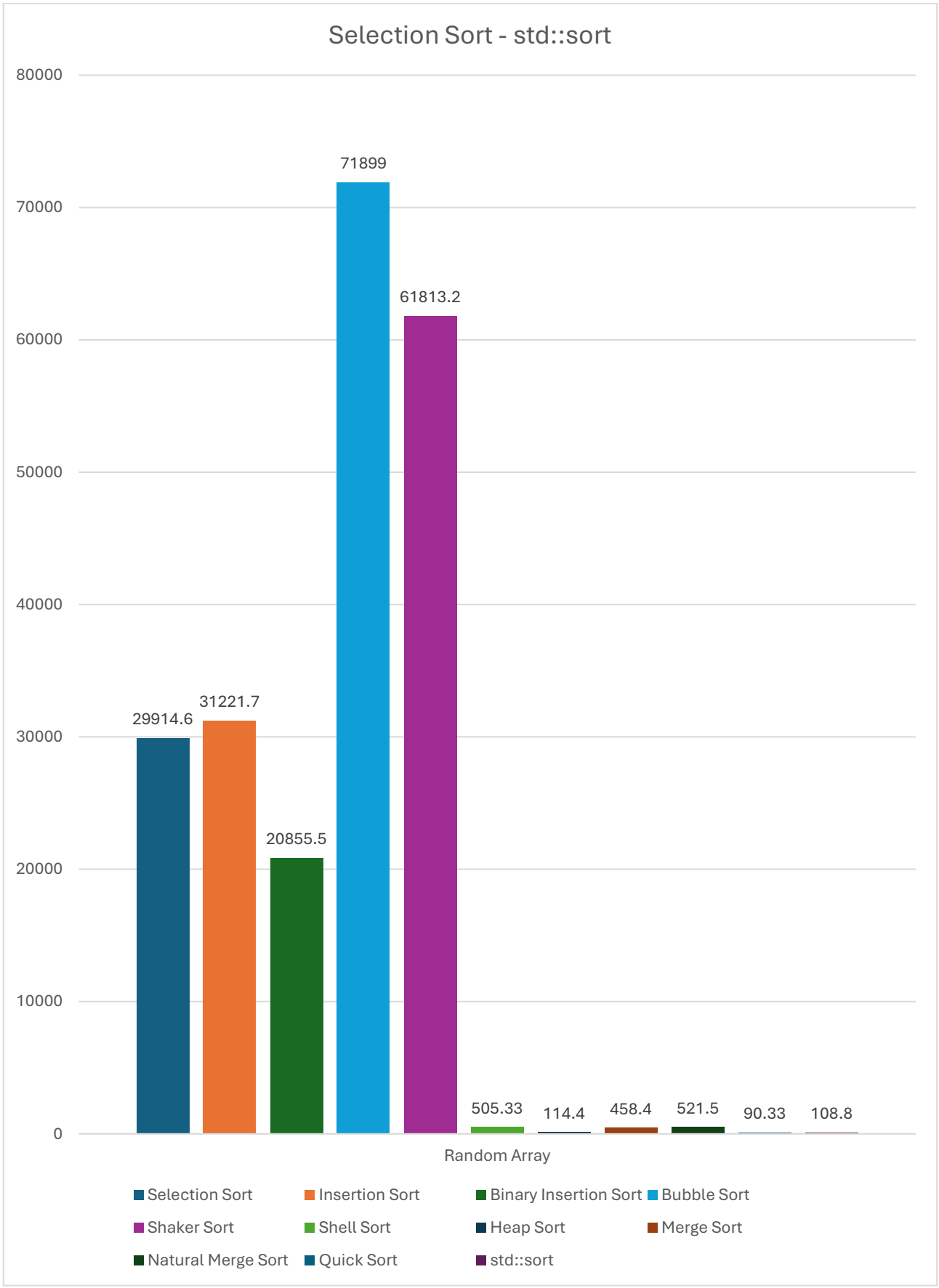

Shell Sort - std::sort
500
450
400
350
300
250
200
150
100
50
0
415.4
110
441.4
417.2
471.6
78.8
Already Sort Array
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort

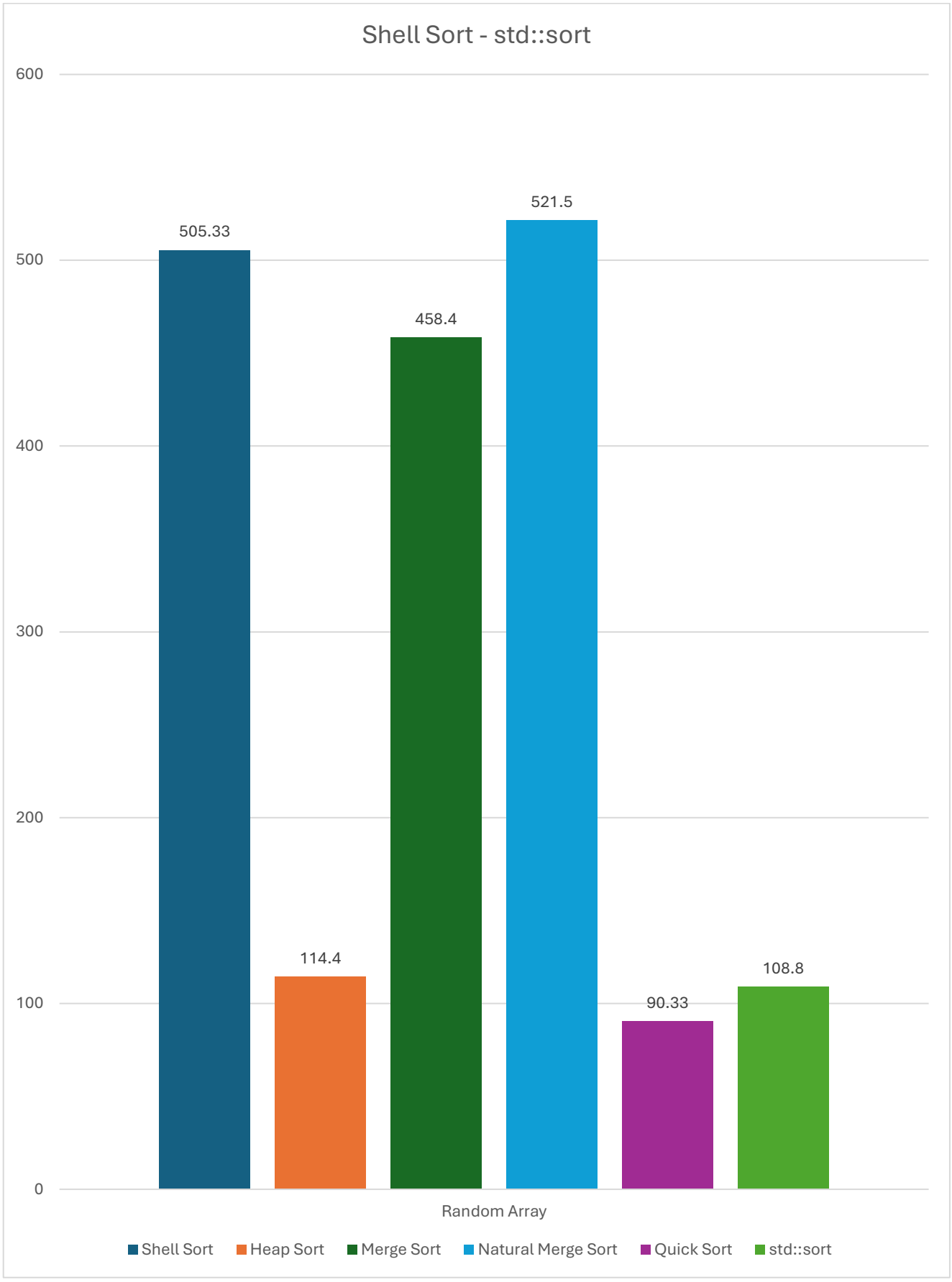

Selection Sort - std::sort
250000
200000
150000
100000
50000
0
37959.4
137271.2
121584
226899
209030.7
953
203.1
908
759
2811.2
93.5
Reverse Sort Array
Selection Sort
Insertion Sort
Binary Insertion Sort
Bubble Sort
Shaker Sort
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort


Shell Sort - std::sort
3000
2500
2000
1500
1000
500
0
953
203.1
908
759
2811.2
93.5
Reverse Sort Array
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort


Selection Sort - std::sort
120000
100000
80000
60000
40000
20000
0
58678.2
18185
11588.2
112555.2
91896.8
771
188.6
811
825.6
167
176.6
Nearly Sort Array
Selection Sort
Insertion Sort
Binary Insertion Sort
Bubble Sort
Shaker Sort
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort


Shell Sort - std::sort
900
800
700
600
500
400
300
200
100
0
771
188.6
811
825.6
167
176.6
Nearly Sort Array
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort


Selection Sort - std::sort
80000
70000
60000
50000
40000
30000
20000
10000
0
29914.6
31221.7
20855.5
71899
61813.2
505.33
114.4
458.4
521.5
90.33
108.8
Random Array
Selection Sort
Insertion Sort
Binary Insertion Sort
Bubble Sort
Shaker Sort
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort


Shell Sort - std::sort
600
500
400
300
200
100
0
505.33
114.4
458.4
521.5
90.33
108.8
Random Array
Shell Sort
Heap Sort
Merge Sort
Natural Merge Sort
Quick Sort
std::sort

3.6. Nhận xét

- Nhận xét tổng quát:
  - Các biểu đồ trên không biểu diễn Radix Sort và Counting Sort do 2 thuật toán này có thời gian chạy quá lớn so với các thuật toán còn lại => Không hiệu quả với kiểu dữ liệu string.
  - Bubble Sort luôn cho thời gian chạy vô cùng lớn.
  - std::sort chạy khá hiệu quả với thời gian vô cùng bé so với các thuật toán khác.
- Nhận xét riêng:
  - Random Array:
    - Bubble và Shaker Sort là 2 thuật toán chạy chậm nhất.
    - Heap Sort, Quick Sort và std::sort cho thời gina chạy nhanh nhất tương tự Reverse Sort Array.
  - Already Sorted Array:
    - Selection Sort, Shaker Sort và Bubble Sort cho thời gian chạy khá lớn, rõ rệt so với các thuật toán còn lại.
    - Heap Sort và std:: sort chạy khá nhanh, thời gian chạy khoảng bằng 0.3% so với 3 thuật toán trên.
    - Các thuật toán còn lại chạy với thời gian tương đồng nhau, không chênh lệch nhiều và tương đối nhanh.
  - Reverse Sorted Array:
    - Bubble Sort và Shaker chạy với thời gian khá lớn so với các thuật toán còn lại.
    - Binary và Insertion Sort cho thời gian chạy ở mức tầm trung, bằng một nửa thời gian 2 thuật toán trên.
    - Các thuật toán từ Shell Sort tới std::sort chạy với thời gian bé, trong đó:
      - Quick Sort chạy lâu nhất trong các thuật toán này.
      - std::sort cho thời gian chạy vô cùng bé.
  - Nearly Sorted Array:

- Shaker Sort và Bubble Sort vẫn cho kết quả chạy kém hiệu quả so với các thuật toán còn lại.
- Trong các thuật toán từ Shell Sort đến std::sort, Shell – Merge – Natural Merge Sort cho thời gian chạy tương đồng nhau, trong khi đó 3 thuật toán còn lại chạy nhanh nhất.

TỔNG KẾT

- **Nhóm độ phức tạp** $O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort, Binary Insertion Sort, Shaker Sort
- **Nhóm** $O(n \log n)$: Quick Sort, Heap Sort, Merge Sort, Natural Merge Sort, Shell Sort, `std::sort`
- **Nhóm** $O(n)$: Radix Sort, Counting Sort

Mục tiêu đạt được

- Cài đặt chính xác các thuật toán và kiểm tra tính đúng đắn.
- Thực hiện đo thời gian thực thi các thuật toán trên nhiều loại dữ liệu: Random, Already Sorted, Reverse Sorted, Nearly Sorted Array.
- Trực quan hóa kết quả qua bảng số liệu và biểu đồ.

Kết quả và phân tích

- **Ảnh hưởng của kích thước mảng:**
  - Với mảng nhỏ ($n \leq 20.10^4$), các thuật toán đơn giản vẫn có thể sử dụng được.
  - Khi kích thước mảng lớn ($n \geq 10^6$), các thuật toán $O(n^2)$ trở nên kém hiệu quả rõ rệt.
- **Ảnh hưởng của loại dữ liệu đầu vào:**
  - *Mảng đã sắp xếp*: Insertion Sort, Binary Insertion Sort cho hiệu năng gần $O(n)$.
  - *Mảng gần sắp xếp*: Natural Merge Sort và Binary Insertion Sort hoạt động hiệu quả.
  - *Mảng ngẫu nhiên*: Quick Sort và `std::sort` hoạt động nhanh và ổn định.
  - *Mảng đảo ngược*: Counting Sort hiệu quả nếu phạm vi giá trị nhỏ.
- **So sánh tổng quát:**

- Quick Sort và `std::sort` luôn nằm trong top thuật toán có hiệu suất cao.
- Counting Sort hoạt động nhanh nhưng tiêu tốn bộ nhớ và kém hiệu quả nếu phạm vi giá trị lớn.
- Radix Sort nhanh với số nguyên nhưng hạn chế về kiểu dữ liệu.

Kết luận tổng quát

Không có một thuật toán nào là tối ưu tuyệt đối cho mọi trường hợp. Việc lựa chọn thuật toán phù hợp phụ thuộc vào:

- Kích thước dữ liệu
- Tính chất và cấu trúc dữ liệu đầu vào
- Hạn chế về tài nguyên hệ thống (thời gian, bộ nhớ, độ ổn định)