TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG TP.HCM

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CÁC GIẢI THUẬT SẮP XẾP

Giáo viên hướng dẫn: Trần Hoàng Quân

Họ và tên	Mã số sinh viên	Lớp
Hoàng Ngọc	23120146	23 CTT2
Tô Thành Long	23120143	23 CTT2
Nguyễn Thanh Phong	23120154	23 CTT2
Tống Thanh Phúc	23120158	23 CTT2

TP. Hồ Chí Minh, 2024

MỤC LỤC

PHỤ LỤC

1. Thông tin nhóm

Họ và tên	Mã số sinh viên	Lớp	Email
Hoàng Ngọc	23120146	23CTT2	23120146@student.hcmus.edu.vn
Tô Thành Long	23120143	23CTT2	23120143@student.hcmus.edu.vn
Nguyễn Thanh Phong	23120154	23CTT2	23120154@student.hcmus.edu.vn
Tống Thanh Phúc	23120158	23CTT2	23120158@student.hcmus.edu.vn

2. Giới thiệu

Các thuật toán sắp xếp là một trong những khái niệm cốt lõi của khoa học máy tính, đóng vai trò nền tảng trong việc xử lý và tổ chức dữ liệu. Chúng cho phép sắp xếp các phần tử trong một tập hợp theo thứ tự nhất định, thường là tăng dần hoặc giảm dần, từ đó giúp tối ưu hóa các thao tác như tìm kiếm, phân loại và phân tích dữ liệu. Việc hiểu rõ nguyên lý hoạt động, ưu nhược điểm và độ phức tạp của từng thuật toán là điều cần thiết để lựa chọn phương pháp sắp xếp phù hợp với từng tình huống cụ thể.

Các thuật toán sắp xếp không chỉ khác nhau về mặt logic hoạt động mà còn có những đặc điểm riêng về hiệu suất trong các trường hợp dữ liệu khác nhau. Một số thuật toán đơn giản nhưng kém hiệu quả khi xử lý tập dữ liệu lớn, trong khi những thuật toán phức tạp hơn lại cho thấy ưu thế vượt trội về mặt thời gian và tài nguyên sử dụng. Chính vì vậy, việc nghiên cứu và so sánh các thuật toán là vô cùng quan trọng trong quá trình thiết kế và tối ưu hóa các hệ thống phần mềm.

Một số thuật toán được nghiên cứu trong báo cáo:

- Insertion Sort
- Bubble Sort
- Merge Sort
- Radix Sort
- Flash Sort
- Selection Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Counting Sort
- Shaker Sort

Báo cáo này sẽ tập trung phân tích chi tiết các thuật toán sắp xếp phổ biến, bao gồm giải thích nguyên lý hoạt động, độ phức tạp thời gian và không gian, cùng với các ví dụ minh họa cụ thể. Bên cạnh đó, các kết quả thử nghiệm thực tế trên nhiều dạng dữ liệu khác nhau cũng sẽ được trình bày để cung cấp cái nhìn trực quan hơn về hiệu suất của từng thuật toán.

Từ đó, người đọc sẽ có đủ cơ sở để lựa chọn và áp dụng thuật toán sắp xếp phù hợp nhất cho bài toán của mình.

3. Các thuật toán khảo sát

Đối với mỗi thuật toán, chúng ta sẽ phân tích các yếu tố:

- **Ý tưởng thuật toán:** Đây là phần giải thích khái niệm cơ bản và phương pháp chính mà thuật toán sử dụng để giải quyết vấn đề. Phần ý tưởng cung cấp cái nhìn tổng quan về cách tiếp cận thuật toán, cách thuật toán ấy hoạt động trong các trường hợp khác nhau và nó hiệu quả trong những trường hợp nào.
- **Các bước của thuật toán:** Đây là phần trình bày chi tiết các bước mà thuật toán thực hiện để đạt được kết quả. Mục đích là giúp người đọc hiểu được cách thức vận hành của thuật toán đó.
- **Mã giả:** Đây là phần mô phỏng các bước của thuật toán bằng các sử dụng từ ngữ theo phong cách ngôn ngữ lập trình nhưng với cú pháp đơn giản và dễ hiểu hơn. Mã giả giúp người đọc tiếp cận gần hơn với những dòng mã nguồn thực tế của thuật toán.
- **Minh hoạ thuật toán:** Phần này sử dụng các hình khối đơn giản để minh hoạ từng giai đoạn của quá trình chạy thuật toán, giúp người đọc hình dung nhanh và hiệu quả hơn khi tiếp cận.
- **Đánh giá độ phức tạp thời gian:** Phần này sẽ phân tích các trường hợp khác nhau bao gồm *best case, average case và worst case* và khi nào sẽ xảy ra từng trường hợp đó.
- **Đánh giá độ phức tạp không gian:** Phần này sẽ trình bày về các thành phần ảnh hưởng đến việc tạo bộ nhớ thêm khi thuật toán được thực thi (mảng phụ,…).
- **Cải tiến thuật toán:** Phần này đề cập đến các cải tiến có thể được áp dụng để làm cho thuật toán hiệu quả hơn về thời gian hoặc không gian. Mục tiêu là giảm thiểu chi phí tính toán hoặc bộ nhớ mà thuật toán cần để hoạt động.

3.1 Insertion Sort

3.1.1 Ý tưởng

Thuật toán sắp xếp Insertion Sort chia mảng thành hai phần, một phần chứa các giá trị đã được sắp xếp và phần còn lại chứa các giá trị chưa được sắp xếp. Thuật toán liên tục lấy một giá trị từ phần chưa sắp xếp và chèn nó vào vị trí đúng trong phần đã sắp xếp, lặp lại quá trình này cho đến khi toàn bộ mảng được sắp xếp. [1]

3.1.2 Các bước của thuật toán

- **Bước 1:** Bắt đầu với việc chọn phần tử thứ hai trong mảng làm khóa, xem như phần tử đầu tiên đã được sắp xếp.
- **Bước 2:** So sánh khóa với các phần tử trong phần đã sắp xếp. Dịch chuyển bất kỳ phần tử nào lớn hơn khóa sang vị trí bên phải để tạo khoảng trống cho khóa.
- **Bước 3:** Đặt khóa vào vị trí đúng của nó trong phần đã sắp xếp của mảng. Lặp lại với tất cả các phần tử.
- **Bước 4:** Tiếp tục quá trình này với từng phần tử chưa được sắp xếp còn lại cho đến khi toàn bộ mảng được sắp xếp.

3.1.3 Mã giả

INSERTION SORT

```
# INSERTION SORT (A, n)
# input: an array A of n elements
# output: an array in ascending order
for i = 1 to n - 1
   do key = A[i]
      j = i - 1
      while j >= 0 and A[j] > key
      do A[j + 1] = A[j]
         j = j - 1;
   A[j + 1] = key
```

3.1.4 Minh hoạ thuật toán

[2]

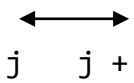
Initially:

12	8	10	2	6	4

First pass:

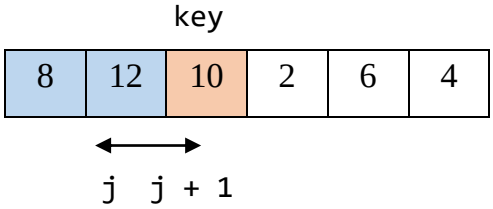
key

12	8	10	2	6	4



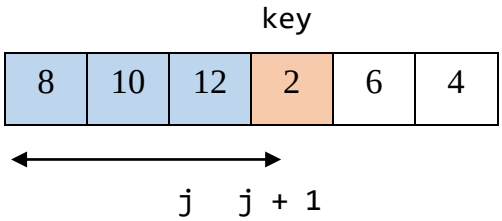
8 < 12, hoán đổi 8 và 12

Second pass:



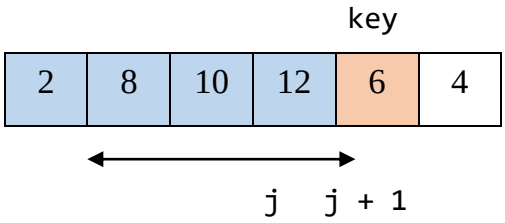
10 < 12, di chuyển 12 sang phải, 10 lớn hơn 8 nên dừng lại

Third pass:



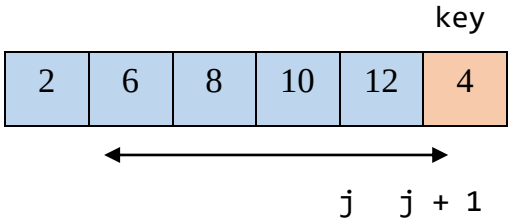
2 đều nhỏ hơn 12, 10, 8 nên chuyển tất cả sang phải, đưa 2 lên đầu

Fourth pass:



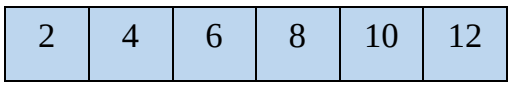
6 đều nhỏ hơn 12, 10, 8 nên chuyển tất cả sang phải, chèn 6 vào đúng vị trí.

Fifth pass:



4 đều nhỏ hơn 12, 10, 8, 6 nên chuyển tất cả sang phải, chèn 4 vào đúng vị trí.

Last pass:



3.1.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian:

- **Best case:** $O(n)$, khi danh sách (mảng) đã được sắp xếp, với n là số phần tử trong danh sách (mảng).
- **Average case:** $O(n^2)$, khi danh sách (mảng) được tạo ngẫu nhiên.
- **Worst case:** $O(n^2)$, khi danh sách (mảng) bị đảo ngược.

Độ phức tạp không gian: `O(1).` Thuật toán Insertion Sort chỉ yêu cầu không gian bổ trợ (Auxiliary Space), nghĩa là không cần thêm nhiều bộ nhớ ngoài, làm cho nó trở thành một thuật toán sắp xếp hiệu quả về mặt không gian.

3.1.6 Cải tiến của thuật toán

Fast Insertion Sort [12]

- **Mô tả:** Ý tưởng chính của thuật toán là trong mỗi lần lặp, phần bên trái của mảng được mở rộng bằng cách chèn một khối gồm k phần tử đã được sắp xếp (với $k \geq 2$) thay vì từng phần tử riêng lẻ như Insertion sort. Thuật toán mới này là một chuỗi thuật toán lồng nhau, Mỗi thuật toán trong chuỗi giải quyết vấn đề bằng cách gọi đệ quy đến các thuật toán trước đó trong cùng một chuỗi.
- **Cải tiến:**
  - Độ phức tạp thời gian: Trong trường hợp xấu nhất là $O\left(n^{1+\frac{1}{h}}\right), với\ h\ \in N$, so với $O(n^2)$ của Insertion sort.
  - Ưu điểm: Fast Insertion Sort có thời gian thực nghiệm là $O(nlogn)$, vượt trội hơn một số thuật toán khác như Merge Sort và Heapsort trên các mảng dữ liệu lớn bất kể dữ liệu đã sắp xếp một phần hay chưa.
  - Độ phức tạp không gian: yêu cầu thêm $O(k)$ không gian bộ nhớ phụ, với k là kích thước của khối cần chèn. Bằng cách thực hiện thêm một số phép toán, thuật toán có thể được điều chỉnh để thực hiện in-place (tại chỗ).

3.2 Bubble Sort

3.2.1 Ý tưởng

Thuật toán Bubble Sort là một thuật toán sắp xếp cơ bản, hoạt động bằng cách liên tục hoán đổi các phần tử liền kề nếu chúng không đúng thứ tự. Tên gọi "Bubble" xuất phát từ cách thuật toán này hoạt động, khi nó làm cho các giá trị lớn (nhỏ) nhất "nổi lên trên" như bong bóng.

3.2.2 Các bước của thuật toán

- **Bước 1:** Bắt đầu bằng việc lặp qua mảng với từng phần tử một.
- **Bước 2:** Trong mỗi lần lặp, thuật toán sẽ so sánh phần tử hiện tại với phần tử tiếp theo.
- **Bước 3:** Nếu phần tử hiện tại lớn (nhỏ) hơn phần tử tiếp theo, hai phần tử này sẽ được hoán đổi vị trí, đảm bảo rằng phần tử lớn (nhỏ) hơn sẽ được đưa về cuối mảng.

- **Bước 4:** Quá trình này được lặp lại cho đến khi tất cả các phần tử trong mảng được sắp xếp theo thứ tự mong muốn.

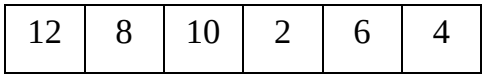
3.2.3 Mã giả

BUBBLE SORT

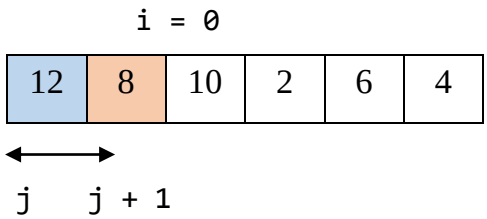
```
# BUBBLE SORT (A, n)
# input: an array A of n elements
# output: an array in ascending order
for i = 0 to n - 2
   for j = 0 to n – 2 - i
      if A[j] > A[j + 1]
         swap (A[j], A[j + 1])
```

3.2.4 Minh hoạ thuật toán

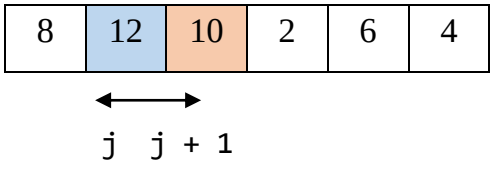
Initially:



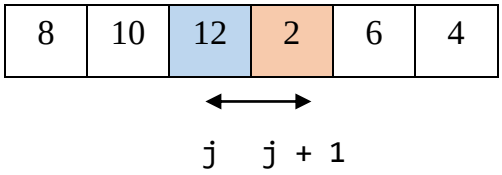
First pass:



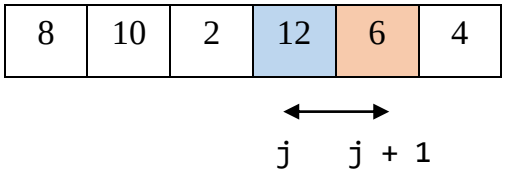
12 > 8, hoán đổi để đưa 8 lên trước



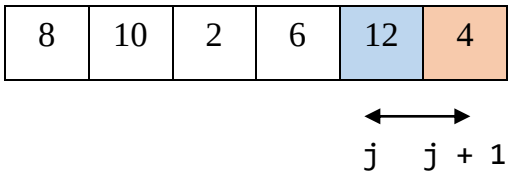
12 > 10, hoán đổi để đưa 10 lên trước



12 > 2, hoán đổi để đưa 2 lên trước

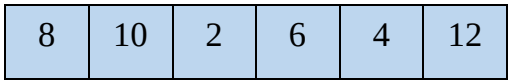


12 > 6, hoán đổi để đưa 6 lên trước

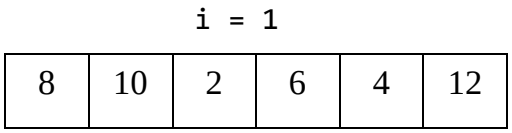


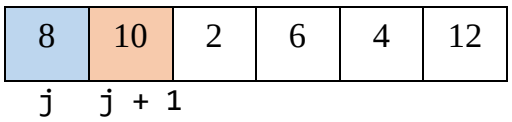
12 > 4, hoán đổi để đưa 4 lên trước

After first pass:

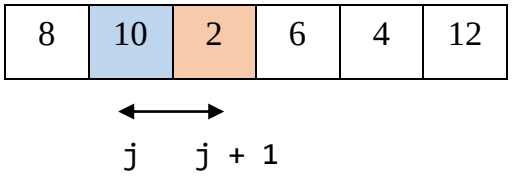


Second pass:

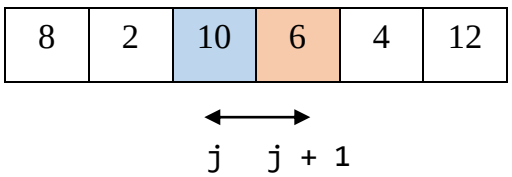




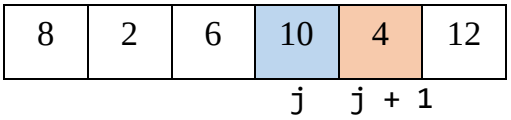
Không cần thay đổi, vì các phần tử 8 và 10 đã được sắp xếp



10 > 2, hoán đổi để đưa 2 lên trước

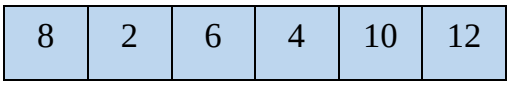


10 > 6, hoán đổi để đưa 6 lên trước



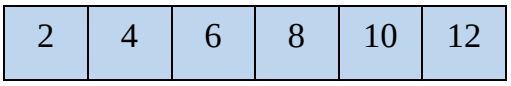
10 > 4, hoán đổi để đưa 4 lên trước

After second pass:



Tiếp tục các vòng lặp tiếp theo với `i = 2, i = 3` để sắp xếp mảng

Last pass:



3.2.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian: [3]

- **Best case:** $O(n^2)$, khi danh sách (mảng) đã được sắp xếp, với n là số phần tử trong danh sách (mảng). Tuy nhiên vẫn có `n - 1` phép so sánh sẽ được thực hiện trong lần lặp đầu tiên, `n - 2` phép so sánh sẽ được thực hiện trong lần lặp thứ hai, `n - 3` phép so sánh sẽ được thực hiện trong lần lặp thứ ba, và cứ tiếp tục như vậy.
  Tổng cộng có tất cả `n*(n - 1) / 2` phép so sánh.
- **Average case:** $O(n^2)$, khi danh sách (mảng) được tạo ngẫu nhiên.
- **Worst case:** $O(n^2)$, khi danh sách (mảng) bị đảo ngược, mọi cặp phần tử đều được so sánh và hoán vị.

Độ phức tạp không gian: $O(1)$. Thuật toán Bubble Sort không cần sử dụng thêm bộ nhớ bổ sung nào ngoài mảng dữ liệu ban đầu.

3.2.6 Cải tiến của thuật toán

Comb Sort [13]

- **Mô tả:** Trong khi Bubble Sort luôn so sánh các giá trị liền kề, Comb Sort sử dụng một khoảng cách ($gap > 1$) giữa hai phần tử được so sánh. Khoảng cách này bắt đầu với giá trị lớn và giảm dần theo hệ số $shrink\ factor = 1.3$ trong mỗi lần lặp cho đến khi nó đạt giá trị 1.
- **Cải tiến:**

- Độ phức tạp thời gian: Trong trường hợp xấu nhất vẫn là $O(n^2)$, xảy ra khi danh sách ban đầu bị đảo ngược hoàn toàn hoặc cần nhiều lần quét để hoàn thành sắp xếp. Trường hợp trung bình và tốt nhất là $O(nlogn)$ nhờ việc giảm gap qua từng lần lặp.
- Ưu điểm: Comb Sort có thể loại bỏ nhiều đảo ngược (inversion) trong một lần hoán đổi, tối ưu cho việc giảm thiểu số lượng so sánh trong các bước đầu của thuật toán, giúp thuật toán hiệu quả hơn Bubble Sort.
- Độ phức tạp không gian: $O(1)$. Thuật toán chỉ sử dụng một số lượng rất ít biến phụ trợ, như các biến để quản lý gap và chỉ số trong quá trình duyệt qua danh sách. Nó không sử dụng bất kỳ cấu trúc dữ liệu bổ sung nào cần bộ nhớ lớn như mảng, stack,...

3.3 Merge Sort

3.3.1 Ý tưởng

Thuật toán Merge Sort là một thuật toán sắp xếp dựa trên giải thuật chia để trị. Thuật toán này hoạt động bằng cách liên tục chia nhỏ mảng đầu vào thành các mảng con nhỏ hơn, sắp xếp từng mảng con này, sau đó gộp chúng lại với nhau để tạo thành mảng đã được sắp xếp hoàn chỉnh.

3.3.2 Các bước của thuật toán

- **Bước 1:** Bắt đầu với việc chia mảng ban đầu thành hai mảng con, mỗi mảng con có kích thước xấp xỉ một nửa mảng gốc.
- **Bước 2:** Tiếp tục chia nhỏ các mảng con theo cách đệ quy, cho đến khi mỗi mảng con chỉ chứa duy nhất một phần tử (được coi là đã sắp xếp).
- **Bước 3:** Bắt đầu quá trình gộp bằng cách so sánh các phần tử từ hai mảng con, đưa phần tử có giá trị nhỏ hơn vào mảng kết quả trước.
- **Bước 4:** Tiếp tục gộp các mảng con lại với nhau theo cách tương tự cho đến khi mảng ban đầu được khôi phục dưới dạng đã sắp xếp.

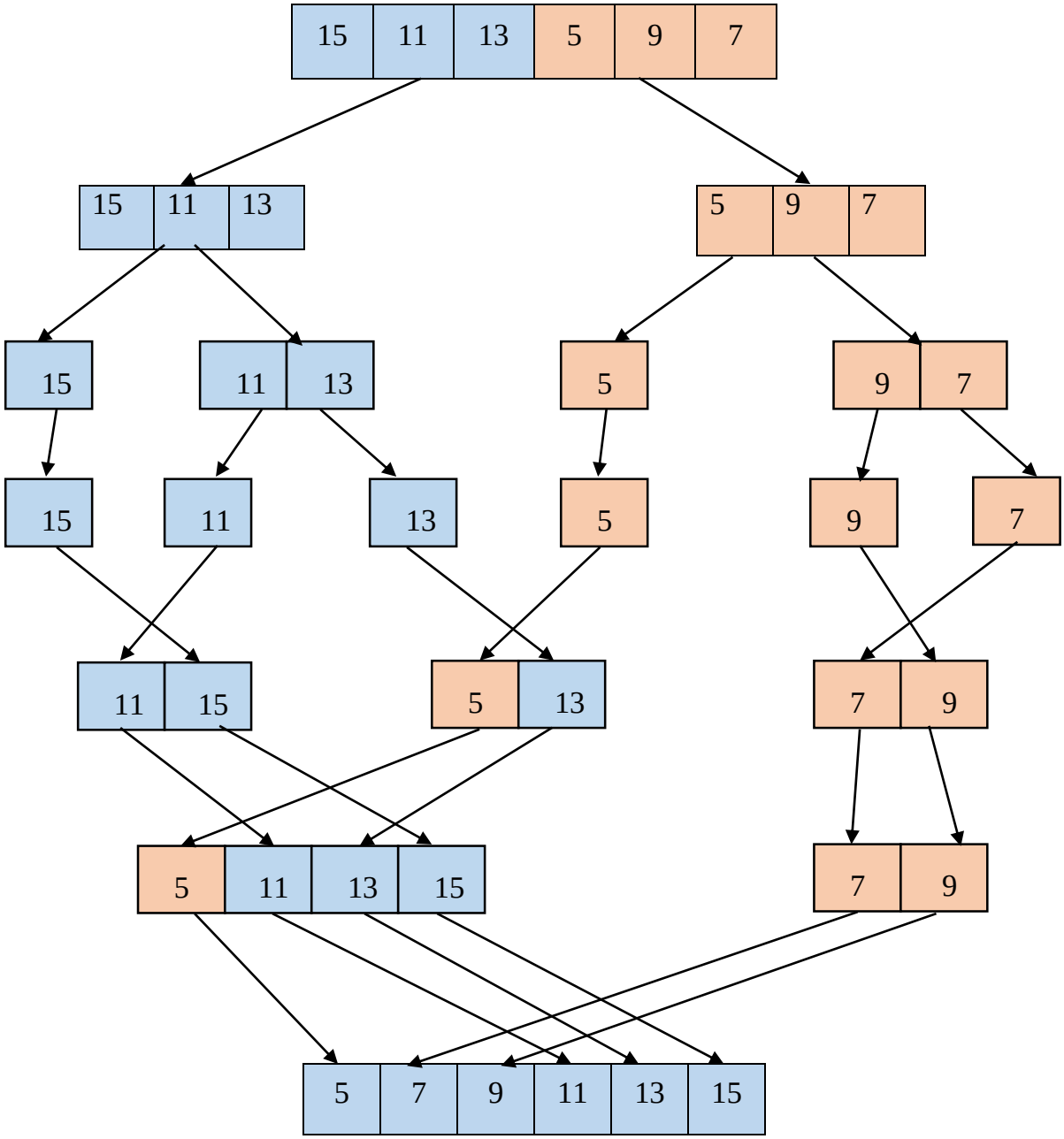
3.3.3 Mã giả

MERGE SORT

```
# MERGE SORT (A, n)
# input: an array A of n elements
# output: an array in ascending order
Function MergeArray (A, n, B, m, C)
   idx1 = idx2 = idx3 = 0
   while idx1 < n and idx2 < m
      if A[idx1] < B[idx2]
         C[idx3++] = A[idx1]
      else C[idx3++] = B[idx2]
   # Copy any leftover elements from left part, if present
   while idx1 < n
      C[idx3++] = A[idx1]
   # Copy any leftover elements from right part, if present
   while idx2 < m
      C[idx3++] = B[idx2]
Function MergeSort (A, left, right)
    if left == right
       temp = new array of size 1
       temp[0] = A[left]
       return temp
    # Split the large array into two smaller parts
    mid = left + (right - left) / 2
    A = MergeSort(arr, left, mid)
    B = MergeSort(arr, mid + 1, right)
    n = mid - left + 1
    m = right – mid
    # Merge 2 sorted parts
    C = new array of size n + m
    MergeArray (A, n, B, m, C)
    Free memory for A and B
    return C
```

3.3.4 Minh hoạ thuật toán

Khởi tạo mảng ban đầu sau đó chia đôi mảng sau mỗi bước, cuối cùng sẽ merge lại để được mảng sắp xếp.



3.3.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian:

- **Best case:** `O(nlogn)`.
- **Average case:** `O(nlogn)`.

- **Worst case:** `O(nlogn).`

Chứng minh [4] [5]

Divide: Mảng kích thước n được chia thành hai mảng con, mỗi mảng có kích thước $n/2$. Việc chia mảng chỉ tốn thời gian hằng số $D(n) = O(1)$.

Conquer: Hai bài toán con được sắp xếp thông qua việc gọi đệ quy, với mỗi bài toán tốn thời gian $T(n/2)$.

Combine: Sau khi sắp xếp, hai mảng con được hợp nhất thành một mảng đã sắp xếp, sử dụng thuật toán gộp với độ phức tạp thời gian $C(n) = O(n)$.

Phương trình đệ quy biểu diễn thời gian chạy của Merge Sort là:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Định lý Master cho phương trình đệ quy trong các thuật toán Divide and Conquer:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Trong đó: a là số lượng bài toán con, b là tỷ lệ thu nhỏ kích thước bài toán con, $f(n)$ là chi phí ngoài đệ quy (như chi phí gộp mảng trong Merge Sort).

So sánh mức tăng trưởng của $n^{\log_b a}$ với $D(n) + C(n)$

1. Nếu $f(n) = D(n) + C(n) = O(n^{\log_b a - \epsilon})$, với $\epsilon > 0$, thì $T(n) = O(n^{\log_b a})$
2. Nếu $f(n) = D(n) + C(n) = O(n^{\log_b a})$, thì $T(n) = O(n^{\log_b a} \log n)$
3. Nếu $f(n) = D(n) + C(n) = W\ (n^{\log_b a + \epsilon})$, với $\epsilon > 0$, thỏa mãn điều kiện thông thường, thì $T(n) = O\big(f(n)\big) = O(D(n) + C(n))$.

Với $a = 2, b = 2$, ta có: $\log_b a = \log_2 2 = 1$.

Theo Định lý Master, ta có:

$f(n) = D(n) + C(n) = O(n) = O(n^{\log_2 2})$, thuộc trờng hợp 2.

Vậy $T(n) = O(n \log n)$.

Độ phức tạp không gian: `O(n).` Thuật toán Merge Sort yêu cầu thêm bộ nhớ không gian `O(n)` vì thuật toán này sử dụng một mảng tạm thời để lưu trữ các kết quả sau khi gộp hai

mảng con lại với nhau. Mảng tạm thời này cần để lưu trữ các giá trị gộp tạm thời trước khi sao chép lại vào mảng ban đầu.

3.3.6 Cải tiến của thuật toán

Parallel Merge Sort [14]

- **Mô tả:** Parallel Merge Sort chia mảng và sắp xếp các mảng con song song bằng cách sử dụng nhiều luồng (threads) hoặc bộ xử lý (processors), sau đó kết hợp kết quả từ các tác vụ song song.
- **Cải tiến:**
  - Độ phức tạp thời gian: `O(logn)`. Ta có n bộ xử lý, thời gian để chia mảng là `logn`, và thời gian để hợp nhất là `logn`. Do đó, tổng thời gian sẽ là `2logn`, tức độ phức tạp trung bình là `O(logn)`.
  - Ưu điểm: Các phép toán có thể được xử lý một cách đồng thời. Điều này giúp giảm đáng kể thời gian thực hiện, đặc biệt là khi xử lý các tập dữ liệu lớn, vì mỗi phần tử trong dữ liệu ban đầu có thể được xử lý nhanh hơn nhờ vào sự phân tán công việc giữa các bộ xử lý, đồng thời giúp sử dụng tối ưu các tài nguyên phần cứng như CPU đa lõi hoặc các hệ thống xử lý phân tán.
  - Độ phức tạp không gian: `O(n)`. Thuật toán vẫn cần phải lưu trữ tất cả các phần tử trong mảng ban đầu, các mảng con trong quá trình chia và hợp nhất và bộ nhớ để quản lý các luồng (thread) và các hàng đợi (queue) khi thực thi song song.

3.4 Radix Sort

3.4.1 Ý tưởng

Thuật toán Radix Sort là thuật toán sắp xếp tuyến tính, sắp xếp các phần tử dựa trên từng chữ số một, từ chữ số ít quan trọng đến chữ số quan trọng nhất. Thay vì so sánh trực tiếp các phần tử, nó phân phối các phần tử vào các bucket dựa trên giá trị của từng chữ số, giúp đạt được thứ tự sắp xếp cuối cùng.

3.4.2 Các bước của thuật toán

- **Bước 1:** Bắt đầu từ chữ số ít quan trọng nhất (chữ số ở vị trí cuối cùng), tiến hành phân loại các phần tử vào các nhóm dựa trên giá trị của chữ số này.
- **Bước 2:** Sắp xếp các phần tử trong từng nhóm, sau đó phục hồi thứ tự vào mảng ban đầu theo thứ tự nhóm, duy trì tính sắp xếp của các chữ số trước đó.

- **Bước 3:** Tiến hành lặp lại quá trình trên cho các chữ số có độ quan trọng cao hơn (từ trái qua phải), cho đến khi tất cả các chữ số đã được xử lý xong. [7]

3.4.3 Mã giả

RADIX SORT

```
# RADIX SORT (A, n)
# input: an array A of n elements
# output: an array in ascending order
Function getMax (A, n) # find the maximum element
   maxi = A[0]
   for i = 1 to n - 1
      if A[i] > maxi
         maxi = A[i]
Function countSort (A, n, exp) [6]
   create output[n]
   // create count array of size 10 initialized to 0
   for i = 0 to n - 1
     count[(A[i] / exp) % 10]++
   for i = 1 to 9
     count[i] = count[i] + count[i - 1]
   for i = n - 1 to 0
     output[count[(arr[i] / exp) % 10] - 1] = A[i]
     count[(A[i] / exp) % 10]--
   for i = 0 to n - 1
     A[i] = output[i]
Function radixSort(A, n)
   maxi = getMax(A, n)
   for exp = 1 to maxi, exp = exp * 10
   countSort(A, n, exp)
```

3.4.4 Minh hoạ thuật toán

Initially:

1912	736	2000	280	517	785

First pass:

200**0**	28**0**	191**2**	78**5**	73**6**	51**7**

Sắp xếp tăng dần theo hàng đơn vị

Second pass:

20**0**0	19**1**2	5**1**7	7**3**6	2**8**0	7**8**5

Sắp xếp tăng dần theo hàng chục

Third pass:

2**0**00	**2**80	**5**17	**7**36	**7**85	1**9**12

Sắp xếp tăng dần theo hàng trăm

Fourth pass:

0280	**0**517	**0**736	**0**785	**1**912	**2**000

Sắp xếp tăng dần theo hàng ngàn

Last pass:

280	517	736	785	1912	2000

3.4.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian: [8]

- **Best case:** `O(a(n + b))`, khi tất cả các phần tử đều có cùng số chữ số. Trong đó a là số lần quét qua dữ liệu b là số chữ số tối đa của phần tử lớn nhất.
- **Average case:** `O(a(n + b))`, khi danh sách (mảng) được tạo ngẫu nhiên. Trong đó a là số lần quét qua dữ liệu b là số chữ số tối đa của phần tử lớn nhất.
- **Worst case:** $O(n^2)$, khi tất cả các phần tử đều có cùng số chữ số. Nếu số chữ số của phần tử lớn nhất là n thì độ phức tạp thời gian là $O(n^2)$.

Độ phức tạp không gian: `O(n + k)`.Thuật toán Radix Sort sử dụng Counting Sort, thuật toán này cần sử dụng các mảng phụ có kích thước n và k. Trong đó n là số lượng phần tử

trong mảng đầu vào, k là giá trị lớn nhất trong các chữ số ở vị trí thứ d (hàng đơn vị, hàng chục, hàng trăm,...) của mảng đầu vào.

3.4.6 Cải tiến của thuật toán

Parallel Radix Sort

- **Mô tả:** Parallel Radix Sort sử dụng nhiều bộ xử lý hoặc luồng để xử lý các chữ số khác nhau của dữ liệu đồng thời. Điều này giúp tăng tốc độ sắp xếp, đặc biệt là với các tập dữ liệu lớn.
- **Cải tiến:**
  - Độ phức tạp thời gian: Trung bình là `O(a(n + b))` tương tự Radix Sort thông thường.
  - Ưu điểm: Các phép toán có thể được xử lý một cách đồng thời. Điều này giúp giảm đáng kể thời gian thực hiện, đặc biệt là khi xử lý các tập dữ liệu lớn (có nhiều chữ số), vì mỗi phần tử trong dữ liệu ban đầu có thể được xử lý nhanh hơn nhờ vào sự phân tán công việc giữa các bộ xử lý.
  - Độ phức tạp không gian: `O(1)`.Thuật toán không cần các bước gọi đệ quy nên không sử dụng các cấu trúc dữ liệu bổ sung lớn như mảng con hay stack để quản lý. Chỉ cần bộ đếm hoặc mảng nhỏ để quản lý quá trình sắp xếp và hợp nhất.

3.5 Flash Sort

3.5.1 Ý tưởng

Thuật toán Flash Sort hoạt động bằng cách phân loại dài hạn (Long-range ordering), ánh xạ các phần tử vào các lớp dựa trên giá trị của chúng và thực hiện hoán vị tại chỗ để đưa phần tử về gần vị trí đúng. Sắp xếp ngắn hạn (Short-range ordering), sử dụng thuật toán so sánh đơn giản để sắp xếp các phần tử trong từng lớp. [9]

3.5.2 Các bước của thuật toán

- **Bước 1:** Classification - Xác định lớp thích hợp cho mỗi phần tử và đếm số lượng phần tử trong mỗi lớp.
- **Bước 2:** Permutation - chuyển dời các phần tử trong mảng về lớp của mình.
- **Bước 2:** Sorting - Sắp xếp lại các phần tử trong mảng dựa trên số lượng phần tử trong mỗi lớp, đảm bảo rằng mỗi phần tử được đặt vào vị trí chính xác của nó. [10]

3.5.3 Mã giả

FLASH SORT [10]

```
# FLASH SORT (A, n)
# input: an array A of n elements
# output: an array in ascending order

Function flashSort(A)
    // Khởi tạo giá trị min, max và các biến cần thiết
    n = size of A
    if n == 0
        return A
    maxIndex = 0
    min = A[0]
    m = floor(0.45 * n)  // Số lớp
    l = A of size m
```

```
// Tìm giá trị min và max trong mảng
for i = 1 to n - 1
    if A[i] < min
        min = A[i]
    if A[i] > A[maxIndex]
        maxIndex = i

// Nếu tất cả các phần tử đều giống nhau, trả về mảng
if min == A[maxIndex]
    return A

// Phân loại với hệ sô phân loại c1
c1 = (m - 1) / (A[maxIndex] - min)

for j = 0 to n - 1
    k = floor(c1 * (A[j] - min))
    l[k] = l[k] + 1

for p = 1 to m - 1
    l[p] = l[p] + l[p - 1]

swap(A[maxIndex], A[0])
// Hoán vị các phần tử để đưa chúng về đúng lớp
move = 0, j = 0, k = m - 1
while move < n - 1
    while j > l[k] - 1
        j = j + 1
        k = floor(c1 * (A[j] - min))
    if k < 0
        break
    flash = A[j]
    while j != l[k]
        k = floor(c1 * (flash - min))
        hold = A[l[k] - 1]
        A[l[k] - 1] = flash
        flash = hold
        l[k] = l[k] - 1
        move = move + 1
// Sử dụng Insertion Sort để sắp xếp mảng hoàn chỉnh
for j = 1 to n - 1
    hold = A[j]
    i = j - 1
    while i >= 0 and A[i] > hold
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = hold
return A
```

3.5.4 Minh hoạ thuật toán

[11]

Số lớp: `m = 0.45 * 7 = 3`

Tìm hệ số phân loại `c1: (m – 1)/(max – min) = (3 – 1)/(81 – 1) = 0.025`

50	77	1	40	11	3	81
Phân lớp: 1	1	0	0	0	0	2

0	1	2
4	2	1

Đếm số phần tử mỗi phân lớp

0	1	2
4	6	7

Cộng dồn để được vị trí kết thúc các phân lớp

Hoán vị `A[max] = 81` với `A[0] = 50`:

81	77	1	40	11	3	50

Tiếp tục lặp lại quá trình duyệt qua các phần tử và đưa chúng vào đúng vị trí trong lớp của chúng, sau đó hoán vị tương tự.

Mảng sau khi đã phân lớp và hoán vị thành công:

11	1	40	3	77	50	81
Phân lớp: 0	0	0	0	1	1	2

Dùng thuật toán Insertion Sort để sắp xếp lại mảng này.

Kết quả cuối cùng:

1	3	11	40	50	77	81

3.5.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian:

- **Best case:** `O(n)`, khi tất cả các phần tử được phân bố đồng đều giữa các lớp.
- **Average case:** `O(n)`, vì giai đoạn phân phối thường đảm bảo rằng các phần tử được phân bố đều giữa các lớp.
- **Worst case:** $O(n^2)$, khi tất cả các phần tử được phân bố không đồng đều giữa các lớp.

Độ phức tạp không gian: `O(n + m)`. Điều này xuất phát từ việc cần thêm bộ nhớ cho mảng đếm các lớp. Trong đó n là số lượng phần tử trong danh sách (mảng), m là số lượng các lớp.

3.5.6 Cải tiến của thuật toán

Hybrid Flash Sort

- **Mô tả:** Ý tưởng chính của thuật toán là tận dụng điểm mạnh của Flash Sort để chia nhỏ mảng ban đầu và sau đó dùng các thuật toán nhanh hơn như Quick Sort để xử lý các mảng con sau khi phân tách.
- **Cải tiến:** Kết hợp ưu điểm của cả hai thuật toán giúp tối ưu hiệu suất sắp xếp. Flash Sort giúp giảm bớt việc so sánh giữa các phần tử trong mảng chính, Quick Sort chỉ cần thực hiện trên các mảng con đã được chia nhỏ.

Parallel Flash Sort

- **Mô tả:** Ý tưởng chính của thuật toán là dữ liệu ban đầu được chia thành nhiều dữ liệu con và có thể được sắp xếp đồng thời bằng cách sử dụng nhiều bộ xử lý hoặc luồng (threads).
- **Cải tiến:** Thời gian sắp xếp tổng thể được cải thiện đáng kể nhờ việc phân phối khối lượng công việc cho nhiều luồng xử lý đồng thời.

3.6 Selection Sort

3.6.1 Ý tưởng

Thuật toán **Selection Sort** là thuật toán sắp xếp dựa trên so sánh. Thuật toán này sắp xếp một mảng bằng cách chọn đi chọn lại phần tử **nhỏ nhất (hoặc lớn nhất)** từ phần chưa được sắp xếp và hoán đổi nó với phần tử chưa được sắp xếp đầu tiên. Quá trình này tiếp tục cho đến khi toàn bộ mảng được sắp xếp.[15]

3.6.2 Các bước của thuật toán

- **Bước 1**: Đầu tiên chúng ta tìm phần tử nhỏ nhất và hoán đổi nó với phần tử đầu tiên. Bằng cách này, chúng ta sẽ có được phần tử nhỏ nhất ở đúng vị trí của nó.
- **Bước 2:** Sau đó, chúng ta tìm phần tử nhỏ nhất trong số các phần tử còn lại (hoặc phần tử nhỏ thứ hai) và hoán đổi nó với phần tử thứ hai.
- **Bước 3:** Chúng ta tiếp tục làm như vậy cho đến khi di chuyển được tất cả các thành phần đến đúng vị trí.

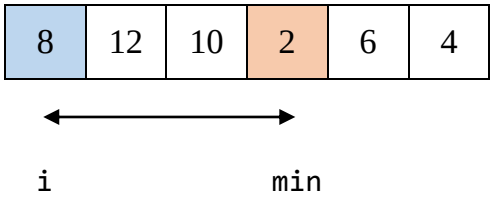
3.6.3 Mã giả

SELECTION SORT

```
# SELECTION SORT (A, n)
# A: an array with n elements
for i = 1 to n - 1
   do min = i
      for j = i+1 to n
          if A[min] > A[j]
          then min = j
   do swap(A[i], A[min])
```

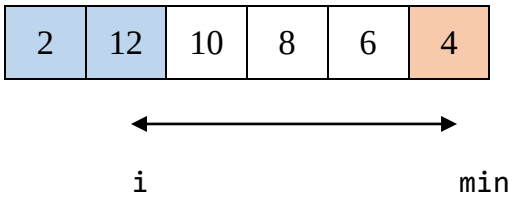
3.6.4 Minh hoạ thuật toán

Initially: 2 is smallest



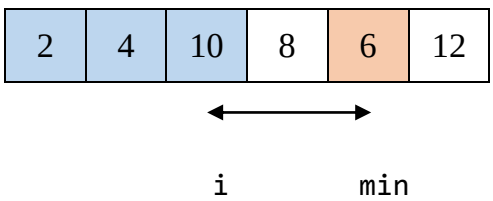
i đang ở vị trí số 8, ta hoán đổi với vị trí min của 2 vì 2<8

First pass: 4 is smallest



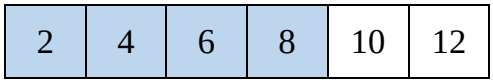
i đang ở vị trí số 12, ta hoán đổi với vị trí min của 4 vì 4<12

Second pass: 6 is smallest



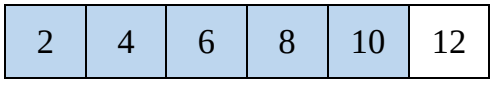
i đang ở vị trí số 10, ta hoán đổi với vị trí min của 6 vì 6<10

Third pass: 8 is smallest



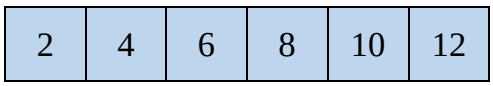
i đang ở vị trí số 8 cũng là vị trí min, do đó không cần hoán đổi

Fourth pass: 10 is smallest



i đang ở vị trí số 10 cũng là vị trí min, do đó không cần hoán đổi

Fifth pass: 12 is smallest



i đang ở vị trí số 12 cũng là vị trí min, do đó không cần hoán đổi

Như vậy, sau khi thuật toán kết thúc, ta đã sắp xếp được dãy tăng dần

3.6.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian

- **Trường hợp tốt nhất (Best case):** $O(n^2)$, luôn so sánh các phần tử bất kể danh sách có được sắp xếp hay chưa, n là số phần tử.
- **Trường hợp trung bình (Average case):** $O(n^2)$, với danh sách được tạo ngẫu nhiên.
- **Trường hợp tệ nhất (Worst case):** $O(n^2)$.

Độ phức tạp không gian: $O(1)$, vì thuật toán sử dụng một lượng bộ nhớ cố định không phụ thuộc vào kích thước của danh sách.

3.6.6 Cải tiến của thuật toán

Stable Selection Sort

- **Mô tả:** Thay vì hoán đổi ở bước 2, phần tử nhỏ nhất được chèn vào vị trí đầu và dịch các phần tử phía sau lên 1 đơn vị.
- **Cải tiến:** Điều này sẽ làm cho thuật toán trở nên ổn định hơn.

Bi-directional Selection Sort

- **Mô tả:** Ở mỗi vòng lặp, ta sẽ tìm cả phần tử lớn nhất và nhỏ nhất, sau đó đưa cả hai về đúng vị trí.
- **Cải tiến:** Điều này sẽ làm giảm số lượng phép so sánh và hoán đổi để sắp xếp.

3.7 Shell Sort

3.7.1 Ý tưởng

Shell Sort là một giải thuật sắp xếp mang lại hiệu quả cao dựa trên giải thuật sắp xếp chèn **(Insertion Sort).** Giải thuật này tránh các trường hợp phải tráo đổi vị trí của hai phần tử xa nhau trong giải thuật sắp xếp chọn (nếu như phần tử nhỏ hơn ở vị trí bên phải khá xa so với phần tử lớn hơn bên trái).[16]

3.7.2 Các bước của thuật toán

- **Bước 1:** Khởi tạo giá trị cho kích thước khoảng cách gọi là **interval**, **interval** sẽ nhận các giá trị lần lượt là `n/2, n/4, n/8`, … cho đến `0.`
- **Bước 2**: Chia mảng thành các mảng con sao cho các số trong mảng có khoảng cách là **interval.**
- **Bước 3:** Sắp xếp các mảng con này bằng thuật toán **Insertion Sort.**
- **Bước 4:** Sau đó, ta giảm interval như Bước 1, và lặp lại Bước 2 cho đến khi interval có giá trị `0.`

3.7.3 Mã giả

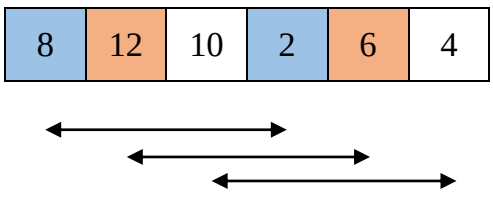
SHELL SORT

```
# SELECTION SORT (A, n)
# A: an array with n elements

//Start with a large gap, then reduce the gap
 gap = n / 2
 while gap > 0 do
    // Sort sub-arrays using Insertion Sort
    for i = gap to n - 1
    do  temp = A[i]
         j = i
         while j >= gap and A[j – gap] > temp
         do A[j] = A[j – gap]
            j = j - gap
         A[j] = temp
     gap = gap / 2
```

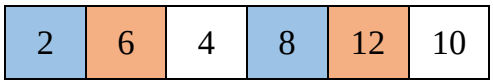
3.7.4 Minh họa thuật toán

First pass: `gap = 6 / 2 = 3`

Ta được các mảng con là `(8, 2), (12, 6), (10, 4),` và ta sẽ thực hiện Insertion Sort cho từng mảng.

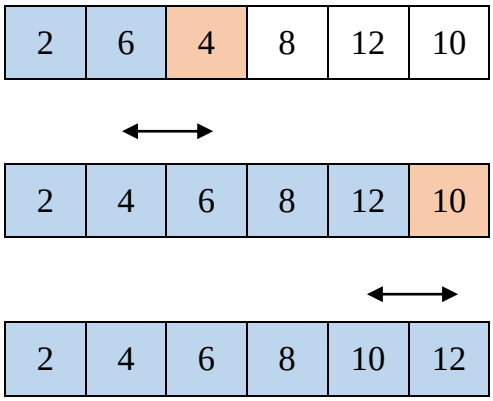


Sau khi thực hiện Insertion Sort cho từng nhóm, đây là kết quả nhận được.



Second pass: `gap = 3 / 2 = 1`

Lúc này, vì `gap=1` nên nhóm lúc này chính là toàn bộ mảng, ta sẽ thực hiện Insertion Sort cho toàn bộ mảng này.



Sau khi thực hiện xong, `gap = 1 / 2 = 0` cũng là lúc mảng đã được sắp xếp

3.7.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian:

- **Trường hợp tốt nhất (Best case):** $O(n\log n)$, khi mảng được sắp xếp, tổng số phép so sánh của mỗi interval sẽ bằng với kích thước mảng đã cho.
- **Trường hợp trung bình (Average case):** $O(n*\sqrt{n})$, với danh sách được tạo ngẫu nhiên.
- **Trường hợp tệ nhất (Worst case):** $O(n^2)$, khi mảng sắp xếp ngược lại so với yêu cầu, khi đó các phép so sánh và hoán đổi là tối đa.

Độ phức tạp không gian: $O(1)$, vì thuật toán chỉ yêu cầu $O(1)$ không gian cho mảng ban đầu. [17]

3.7.6 Cải tiến của thuật toán

Shell Sort có nhiều cải tiến khác nhau, mỗi cải tiến đều ứng với một khoảng cách riêng. Về độ phức tạp về thời gian, các biến thể bên dưới đều là $O(n\log n)$ trong trường hợp tốt nhất và $O(n*\sqrt{n})$ cho trường hợp trung bình. Chúng chỉ khác nhau trong trường hợp xấu nhất.

- Hibbard's sequence ($2^k - 1$, ..., 7, 3, 1): $O(n^{3/2})$. [18]
- Knuth's sequence ($(3^k - 1)/2$, ..., 13, 4, 1): $O(n^{3/2})$. [19]
- Sedgewick's sequence (..., 109, 41, 19, 5, 1): $O(n^{3/2})$. [20]
- Pratt's sequence ($2^i * 3^j$, ..., 6, 3, 2, 1): $O(n\log^2 n)$. [21]

3.8 Heap Sort

3.8.1 Ý tưởng

Heap sort là một kỹ thuật sắp xếp dựa trên so sánh dựa trên Cấu trúc dữ liệu Heap. Sau khi tạo được max-heap (hoặc min-heap), phần tử max (hoặc min) sẽ ở vị trí đầu mảng và ta sẽ hoán đổi nó với phần tử cuối cùng. Chúng ta lặp lại quy trình tương tự cho các phần tử còn lại.[22]

3.8.2 Các bước của thuật toán

- **Bước 1:** Sắp xếp lại các phần tử trong mảng sao cho chúng tạo thành Max-heap.
- **Bước 2:** Hoán đổi phần tử đầu tiên (tức phần tử lớn nhất với heap hiện tại) với phần tử cuối của mảng.
- **Bước 3:** Giảm kích thước heap đi 1 và tiếp tục Heapify lại heap bỏ qua phần tử cuối
- **Bước 4:** Lặp lại bước 2 và 3 cho đến khi chỉ còn một phần tử trong heap.

3.8.3 Mã giả

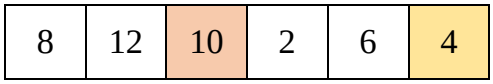
HEAP SORT

```
# HEAP SORT (A, n)
# A: an array with n elements

//Function to Heapify a subtree rooted with node i, which is an index
in A
Heapify(A ,n ,i):
 largest = i
 left = 2 * i + 1
 right = 2 * i + 2
   //If left child is larger than root
    if left < n and A[left] > A[largest]
    then largest = left

   //If right child is larger than largest so far
    if right < n and A[right] > A[largest]
    then largest = right

   //If largest is not root
    if largest != i
    then swap(A[i], A[largest])
    //Recursively heapify the affected sub-tree
         Heapify(A, n, largest)
end Heapify

//Heap sort
for i = n/2 - 1 to 0
   do Heapify(A, n, i)
for i = n - 1 to 0
   do swap(A[0], A[i])
      Heapify(A, i, 0)
```

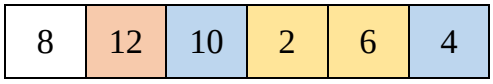
3.8.4 Minh họa thuật toán

Tạo max-heap

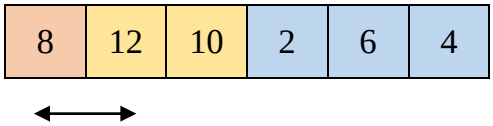
Ta sẽ bắt đầu từ phần tử thứ 6/2 – 1 = 2



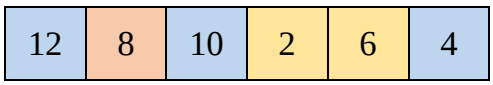
Vì 10 > 4 nên không có hoán đổi nào



Tiếp theo, vì 12 > 2 và 12 > 6 nên không có hoán đổi nào

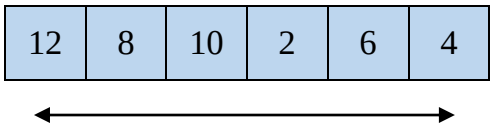


Lúc này, 8 nhỏ hơn con của nó là 12, nên ta hoán đổi vị trí của 8 và 12, đồng thời phải kiểm tra 8 và con của nó



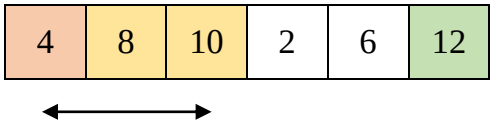
Ta thấy rằng 8 lớn hơn 2 con của nó nên không có sự hoán đổi nào

Sau khi tạo Max-heap, ta sẽ thực hiện các thao tác của Heap-sort



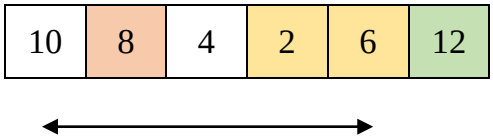
Ta hoán đổi phần tử đầu tiên, cũng chính là phần tử lớn nhất hiện tại với phần tử cuối cùng

First pass:



Sau khi hoán vị, ta cần phải điều chỉnh để đúng với điều kiện

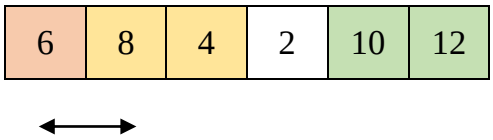
4 nhỏ hơn con của nó là 10, nên ta hoán vị 4 và 10

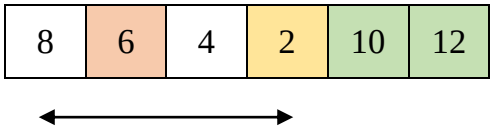


8 lớn hơn 2 con của nó là 2 và 6, do đó không có sự hoán vị nào ở đây

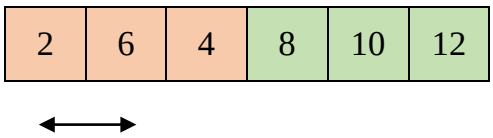
Lúc này, ta tiếp tục hoán vị phần tử lớn nhất hiện tại là 10 và phần tử cuối cùng là 6, và thực hiện các bước điều chỉnh tương tự như trên

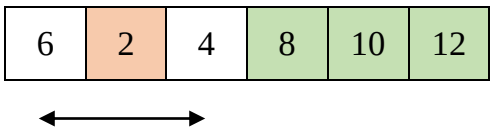
Second pass:



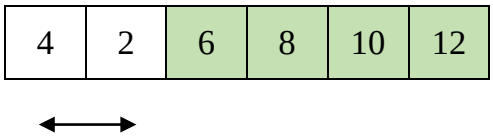


Third pass:

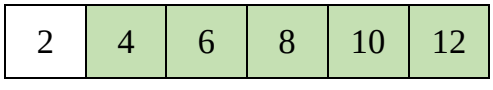




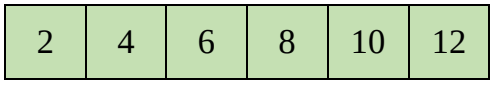
Fourth pass:



Fifth pass:



Sixth pass:



Sau khi thực hiện xong vòng lặp, cũng là lúc mảng đã được sắp xếp thành công

3.8.5 Đánh giá độ phức tạp không gian, thời gian

Độ phức tạp thời gian: Trong mọi trường hợp đều thực hiện các bước giống nhau, độ phức tạp mỗi lần Heapify là `O(logn)` và độ phức tạp của tạo và build heap là `O(n)`, với n là số phần tử của mảng. Do đó, độ phức tạp ở các trường hợp là tương đương.

- **Trường hợp tốt nhất (Best case):** `O(nlogn)`.
- **Trường hợp trung bình (Average case):** `O(nlogn)`.
- **Trường hợp tệ nhất (Worst case):** `O(nlogn)`.

Độ phức tạp không gian: `O(logn)`, khi sử dụng đệ quy cho thao tác Heapify, không gian ngăn xếp phụ thuộc vào chiều cao của heap. Nếu sử dụng vòng lặp thì độ phức tạp có thể là `O(1)`.

3.9 Quick Sort

3.9.1 Ý tưởng

QuickSort là một thuật toán sắp xếp chia để trị (divide-and-conquer). Ý tưởng cơ bản của QuickSort là chọn một phần tử làm "pivot" (chốt) và phân chia mảng sao cho tất cả các phần tử nhỏ hơn pivot sẽ nằm bên trái nó, còn các phần tử lớn hơn pivot sẽ nằm bên phải. Sau đó, thuật toán tiếp tục đệ quy sắp xếp các phần tử bên trái và bên phải của pivot cho đến khi mảng được sắp xếp hoàn toàn.[23]

3.9.2 Các bước của thuật toán

- **Bước 1**: Chọn một phần tử trong mảng (thường là phần tử đầu tiên, cuối cùng hoặc phần tử trung bình)
- **Bước 2:** Sắp xếp lại mảng sao cho tất cả các phần tử nhỏ hơn pivot nằm ở bên trái pivot, và tất cả phần tử lớn hơn pivot nằm ở bên phải pivot.
- **Bước 3:** Áp dụng QuickSort đệ quy cho hai phần con của mảng (bên trái và bên phải pivot).
- **Bước 4:** Khi mảng có một phần tử hoặc không còn phần tử nào, thuật toán dừng lại. [23]

3.9.3 Mã giả

QUICK SORT

```
# QUICK SORT (A, l, r)
# A: A or Subarray of A defined by its left and right index
Quicksort(A, l, r)
   if l < r
   then
     p = Partition(A, l, r)
     Quicksort(A, l, p - 1)
     Quicksort(A, p + 1, r)
end Quicksort
```

- Quá trình chính trong Quick Sort đó chính là việc xây dựng hàm `Partition()`. Mục tiêu của hàm này chính là đưa pivot vào đúng vị trí của nó trong mảng đã được sắp xếp, và đưa các phần tử nhỏ hơn qua phía bên trái, phần tử lớn hơn qua phía bên phải của nó.
- Ở đây, ta sẽ nghiên cứu 2 cách phổ biến:

Phân vùng Hoare: Hoạt động bằng cách khởi tạo hai chỉ mục bắt đầu ở hai đầu, hai chỉ mục di chuyển về phía nhau cho đến khi tìm thấy một nghịch đảo (Giá trị nhỏ hơn ở phía bên trái và giá trị lớn hơn ở phía bên phải). Khi tìm thấy một nghịch đảo, hai giá trị được hoán đổi và quá trình được lặp lại. [24]

QUICK SORT

```
#Return the index of pivot
Partition(A, l, r)
mid = (l + r) / 2
pivot = A[mid]
while l < r
do while A[l] < pivot
   do  l = l + 1
   while A[r] > pivot
   do  r = r – 1
   if l <= r then swap(A[l], A[r])
      l = l + 1 , r = r – 1
swap(A[l], A[r])
return l
```

Phân vùng Lamuto: Thuật toán này hoạt động bằng cách coi phần tử `pivot` là phần tử cuối cùng. Nếu bất kỳ phần tử nào khác được đưa ra làm phần tử `pivot` thì trước tiên hãy hoán đổi nó với phần tử cuối cùng. Bây giờ hãy khởi tạo hai biến `i` là thấp và `j` cũng là thấp, lặp qua mảng và tăng `i` khi `arr[j] <= pivot` và hoán đổi `arr[i]` với `arr[j]` nếu không thì chỉ tăng `j`. Sau khi thoát khỏi vòng lặp, hãy hoán đổi `arr[i+1]` với `arr[hi]`. `i` này lưu trữ phần tử `pivot`.

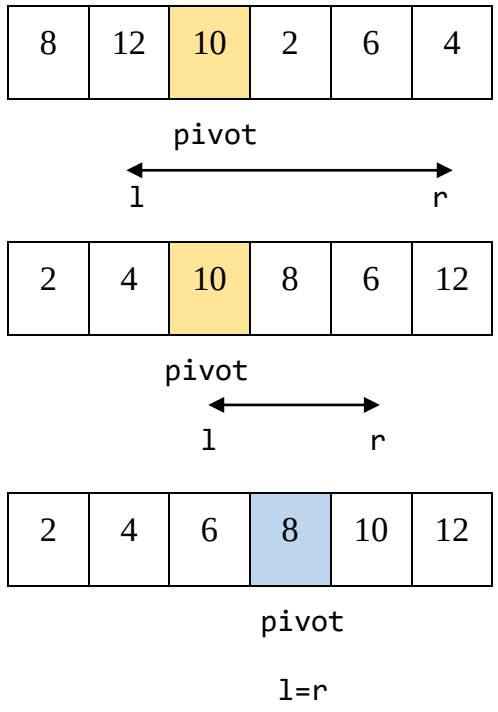
QUICK SORT

```
#Return the index of pivot
Partition(A, l, r)
Pivot = A[r]
i = l
for j = l to r - 1
do   if A[j] <= pivot
     swap(A[i], A[j])
     then i = i + 1
swap(A[i], pivot)
return i
```

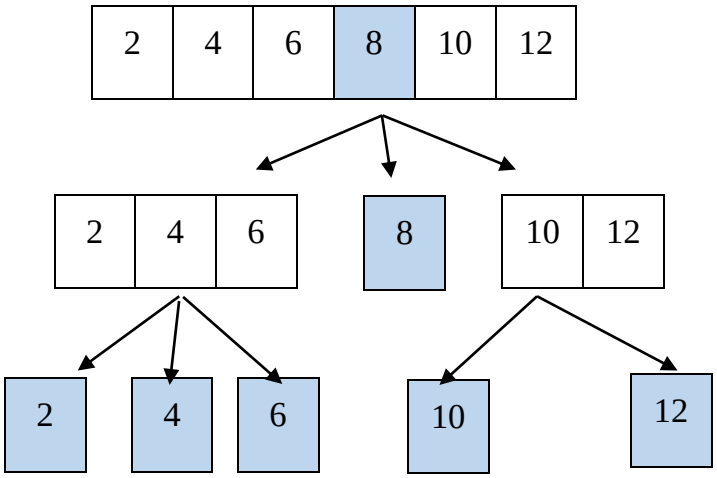
3.9.4 Minh họa của thuật toán

Phân vùng Hoare:

Chỉ số của `pivot = (0 + 5) / 2 = 2`

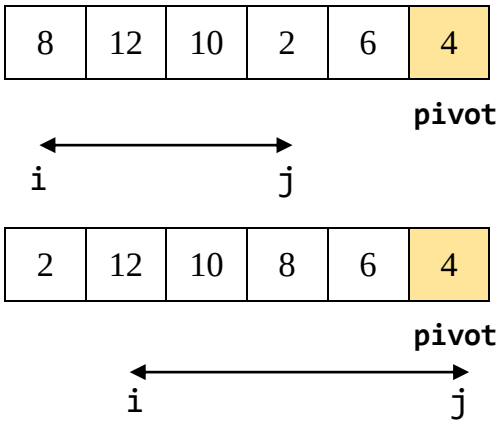


Lúc này, `l` bằng `r`, do đó lần phân vùng dừng lại và `pivot` đã ở đúng vị trí của nó là 8, pivot đã chia mảng ra 2 nửa, ta thực hiện việc phân chia tương tự như trên cho đến khi còn 1 phần tử

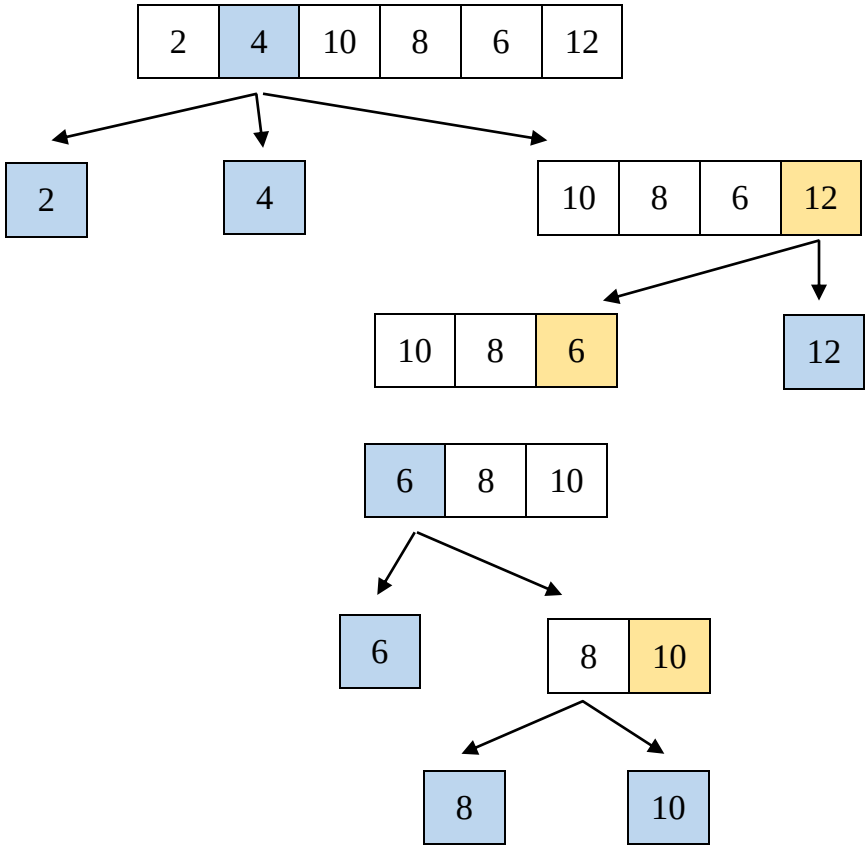


Phân vùng Lomuto:

Chỉ số của `pivot = 5`



Lúc này `j` đã đến phần tử cuối cùng, ta hoán đổi phần tử thứ `i` với `pivot`, khi đó `pivot` đã ở đúng vị trí, `pivot` đã chia mảng ra hai nửa, ta thực hiện việc phân chia tương tự như trên cho đến khi còn 1 phần tử



3.9.5 Đánh giá độ phức tạp thời gian, không gian

Độ phức tạp thời gian:

- **Trường hợp tốt nhất (Best case):** `O(nlogn)`, xảy ra khi phần tử pivot chia mảng thành hai nửa bằng nhau.
- **Trường hợp trung bình (Average case):** `O(nlogn)`, trục chia mảng thành hai phần, nhưng không nhất thiết phải bằng nhau.
- **Trường hợp tệ nhất (Worst case):** $O(n^2)$, xảy ra khi phần tử nhỏ nhất hay lớn nhất luôn được chọn làm vị trí pivot (ví dụ mảng đã được sắp xếp).

Độ phức tạp không gian: `O(n)`, do ngăn xếp gọi đệ quy.

3.9.6. Cải tiến của thuật toán

Randomized Quick Sort

- **Mô tả:** Ý tưởng chính của thuật toán là chọn ngẫu nhiên phần tử chốt (pivot) trong mỗi lần phân đoạn thay vì luôn chọn pivot cố định như phần tử đầu, cuối hoặc giữa mảng.
- **Cải tiến:**

- Độ phức tạp thời gian: Thời gian chạy kỳ vọng là $O(nlogn)$. Thuật toán sử dụng các biến ngẫu nhiên chỉ thị X_{ij} (*indicator random variable*) để đếm số lần các cặp phần tử được so sánh trong quá trình sắp xếp.
- Ưu điểm: Việc chọn pivot ngẫu nhiên giúp giảm thiểu khả năng gặp trường hợp xấu nhất, đặc biệt khi dữ liệu đầu vào đã được sắp xếp hoặc có mẫu nhất định. Điều này đảm bảo rằng, với xác suất cao, thuật toán sẽ hoạt động với hiệu suất trung bình tốt hơn.[30]

3.10 Counting Sort

3.10.1 Ý tưởng

Là thuật toán sắp xếp không dựa trên so sánh . Thuật toán này đặc biệt hiệu quả khi phạm vi giá trị đầu vào nhỏ so với số lượng phần tử cần sắp xếp. Ý tưởng cơ bản đằng sau Counting Sort là đếm tần suất của từng phần tử riêng biệt trong mảng đầu vào và sử dụng thông tin đó để đặt các phần tử vào đúng vị trí đã sắp xếp của chúng. [25]

3.10.2 Các bước của thuật toán

- **Bước 1:** Khai báo một mảng phụ `countArray[]` có kích thước là phần tử lớn nhất trong `inputArray[]` cộng thêm `1` và khởi tạo giá trị bằng `0`.
- **Bước 2:** Duyệt mảng `inputArray[]`, ánh xạ từng phần tử thành chỉ mục của `countArray[]` (tức là `countArray[inputArray[i]]++` với i chạy từ `0` đến `N-1`.
- **Bước 3:** Tính tổng tiền tố tại mọi chỉ mục của mảng `countArray[]` (để khi duyệt, các phần tử đầu vào vào đúng chỉ số ở mảng đầu ra).
- **Bước 4:** Tạo một mảng `outputArray[]` có kích thước `N`.
- **Bước 5:** Duyệt mảng `inputArray[]` từ cuối và gán
  `outputArray[ countArray[ inputArray[i] ] - 1 ]=inputArray[i]`
  `countArray[ inputArray[i] ] = countArray[ inputArray[i] ] –`
- **Bước 6:** Kết quả ở `outputArray[]` chính là `inputArray[]` sau khi được sắp xếp. [26], [27]

3.10.3 Mã giả

COUNTING SORT

```
# COUNTING SORT (A, n)
# A: an array with n elements
// Finding the maximum element of A and Initializing
MaxValue = Max(A, n)
let Count[] be a new array with size m = MaxValue + 1
// Creating count[] from A[]
    for i = 0 to m
    do Count[i] = 0
    for i = 0 to n
    do Count[ A[i] ]++
    for i = 1 to m
    do Count[i] += Count[i - 1]
// Creating outputArray[] from Count[]
let outputArray[] be a new array with size n
for i = n – 1 to 0
do outputArray[Count[ A[i] ] - 1] = A[i]
   Count[ A[i] ]--
return outputArray[]
```

3.10.4 Minh họa của thuật toán

Initially

A[]

8	12	10	2	6	4

Step 1:

`Max = 12,` Tạo một mảng count[] gồm 12 phần tử có giá trị 0

Count[]

0	0	0	0	0	0	0	0	0	0	0	0	0

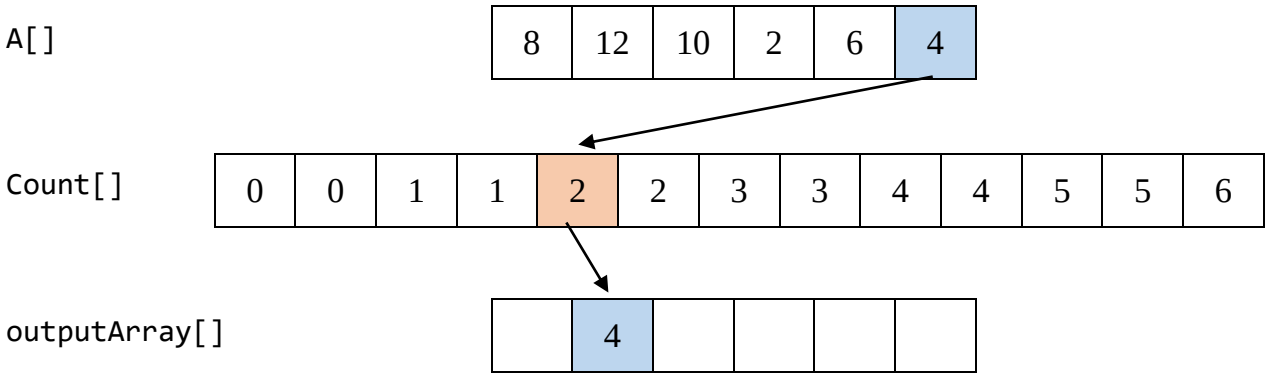
Step 2: Duyệt qua A và tăng các phần tử của mảng Count

Count[]	0	0	1	0	1	0	1	0	1	0	1	0	1

Step 3: Tính tổng tiền tố cho các phần tử

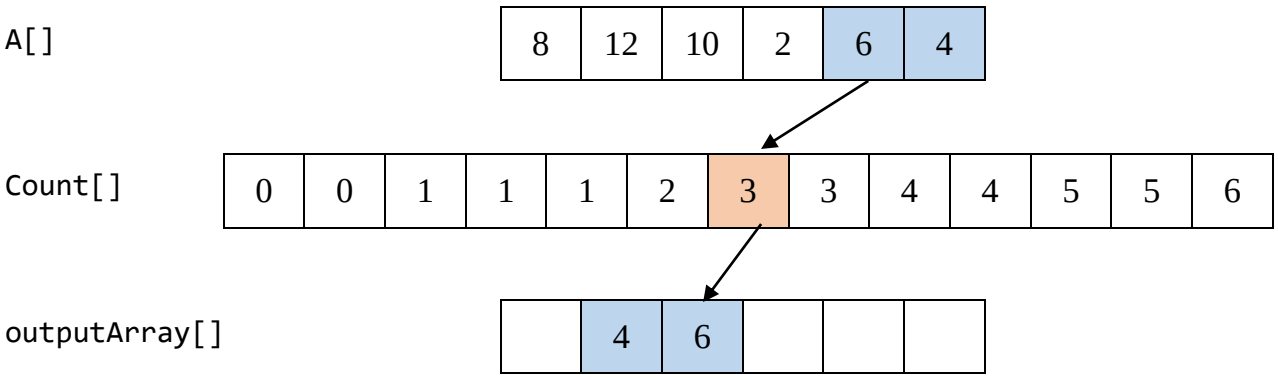
Count[]	0	0	1	1	2	2	3	3	4	4	5	5	6

Step 4: Sau khi đã tạo xong mảng Count, ta sẽ ánh xạ từng phần tử trong A ra đúng vị trí của nó trong outputArray

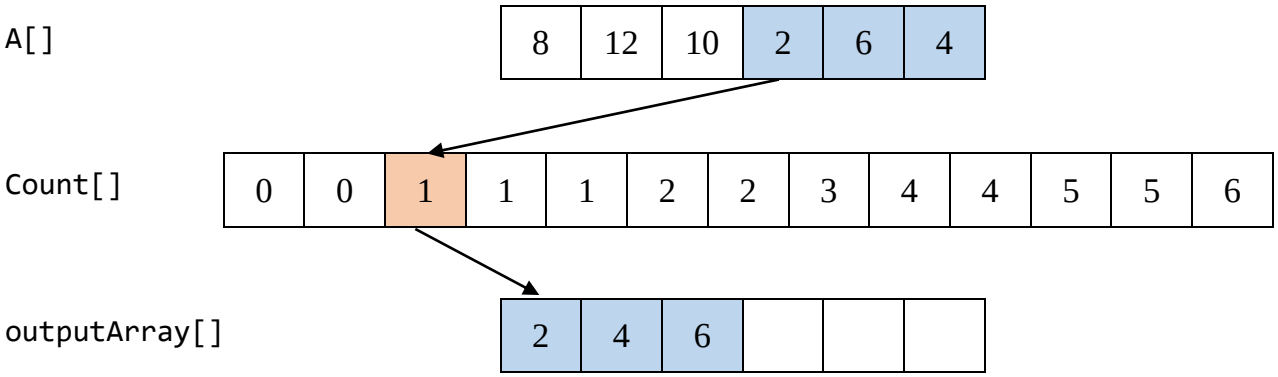


Bắt đầu từ phần tử cuối cùng của A là 4, ánh xạ tới phần tử vị trí 4 của Count là 2, sau đó ánh xạ tiếp tới phần tử thứ 2-1 của outputArray và cho nó bằng 4

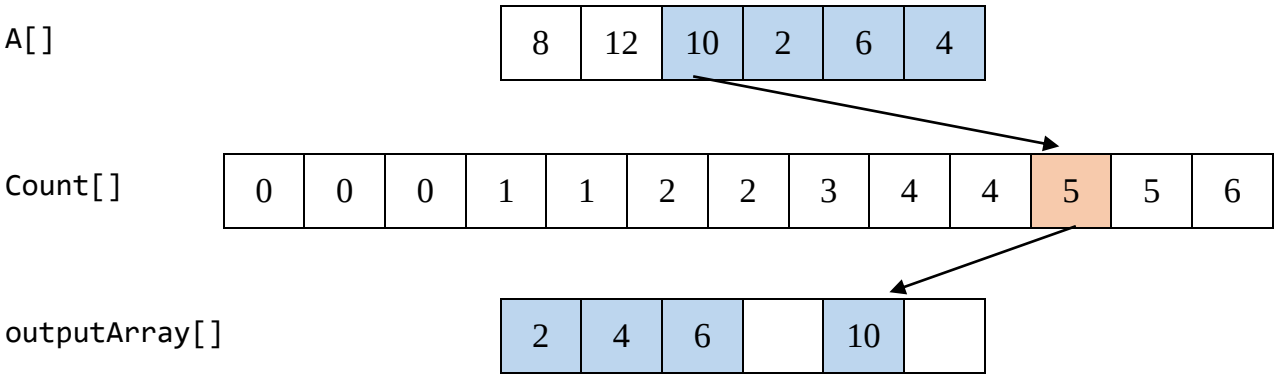
Step 5: Ta thực hiện tương tự như bước trên cho đến khi duyệt qua hết các phần tử của A



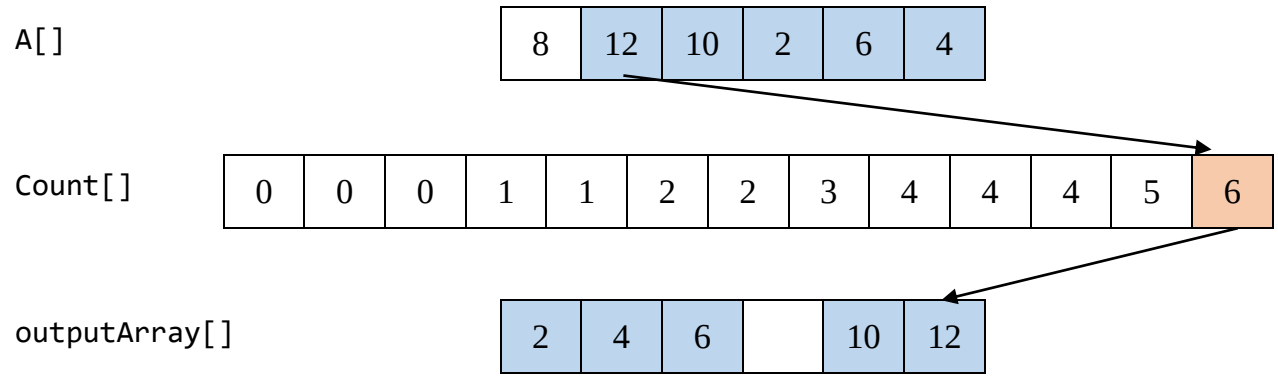
Step 6:



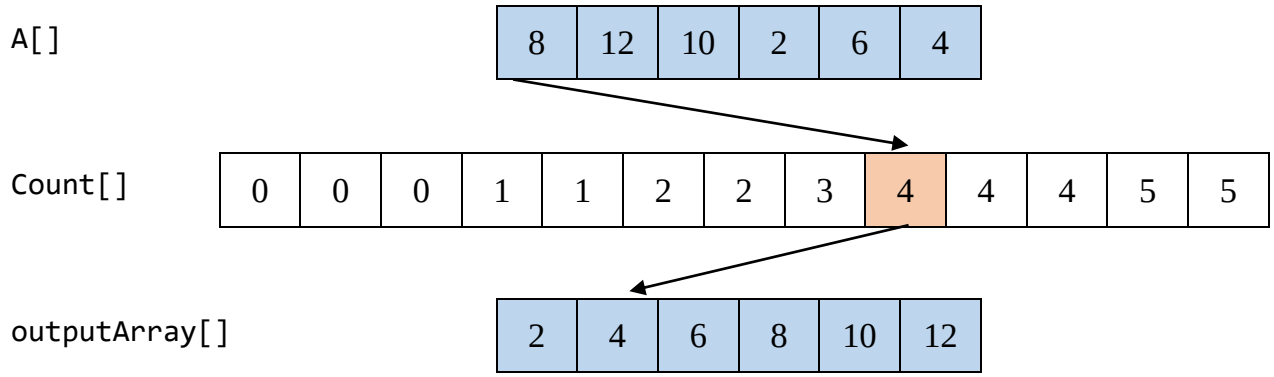
Step 7:



Step 8:



Step 9:



Sau khi thực hiện xong các bước của thuật toán, ta đã được mảng outputArray chính là mảng đã được sắp xếp của A

3.10.5 Đánh giá độ phức tạp của thời gian, không gian

Độ phức tạp thời gian: Trong mọi trường hợp, ta đều phải tạo hai mảng mới và duyệt hai mảng đó. Vì vậy độ phức tạp ở các trường hợp là tương đương

- **Trường hợp tốt nhất (Best case):** `O(n + m)` với `n, m` là kích thước của hai mảng mới được tạo.
- **Trường hợp trung bình (Average case):** `O(n + m).`
- **Trường hợp tệ nhất (Worst case):** `O(n + m).`

Độ phức tạp không gian: `O(n + m),` là không gian được sử dụng bởi hai mảng được tạo.

3.10.6 Cải tiến thuật toán

Radix Sort

- **Mô tả:** Ý tưởng chính của thuật toán là xử lý từng chữ số của số nguyên, bắt đầu từ chữ số ít quan trọng nhất (đơn vị) đến chữ số quan trọng nhất (hàng cao hơn). Thuật toán này thường sử dụng Counting Sort như một bước phụ trợ để sắp xếp các số dựa trên từng chữ số. Đặc biệt Radix Sort là một thuật toán sắp xếp không dựa trên so sánh.
- **Cải tiến:** Hiệu quả với các số nguyên lớn: Bằng cách tập trung vào từng chữ số, Radix Sort có thể sắp xếp các số nguyên lớn một cách hiệu quả, đặc biệt khi kết hợp với Counting Sort để xử lý các chữ số.

3.11 Shaker Sort

3.11.1 Ý tưởng

Là cải tiến của Bubble Sort theo mục tiêu: không chỉ những *phần tử nhẹ nhất* "nổi lên" đầu dãy mà cả *phần tử nặng nhất* cũng "chìm" xuống cuối dãy. Muốn vậy, ta phải ghi nhớ lần đổi chỗ cuối cùng khi duyệt ngược từ cuối lên và khi duyệt xuôi dãy từ đầu xuống cuối để quyết định xem ở bước tiếp theo sẽ duyệt từ đó (chỗ ghi nhớ) đến đâu.[28]

3.11.2 Các bước của thuật toán

- **Bước 1:** Đầu tiên, ta sẽ duyệt mảng từ trái sang phải như **Bubble Sort**. Mỗi lần ta sẽ so sánh hai phần tử liền kề, nếu chúng sai vị trí thì ta hoán đổi. Sau vòng lặp, phần tử lớn nhất ở cuối mảng.
- **Bước 2:** Tiếp theo, ta sẽ duyệt mảng theo hướng ngược lại, bắt đầu từ phần tử đứng trước phần tử lớn nhất vừa được sắp xếp cho đến đầu mảng, hai phần tử liền kề sẽ bị hoán đổi nếu sai vị trí, phần tử nhỏ nhất sẽ nằm đ mảng.
- **Bước 3:** Lặp lại 2 bước trên cho đến khi không còn hai phần tử nào cần được hoán vị trí. [29]

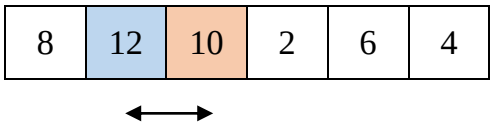
3.11.3 Mã giả

SHAKER SORT

```
# SHAKER SORT (A, n)
# A: an array with n elements
start = 0
end = n - 1
swappable = true
while swappable = true
do swappable = false
   for i = start to end
   do  if A[i] > A[i + 1]
       then swap(A[i], A[i + 1])
            swappable = true
   if swappable = false stop
   swappable = false
   end = end - 1
   for i = end to start
   do  if A[i] > A[i + 1]
       then swap(A[i], A[i + 1])
            swappable = true
   start = start + 1
```

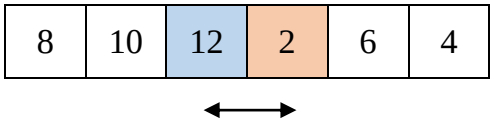
3.11.4 Minh họa của thuật toán

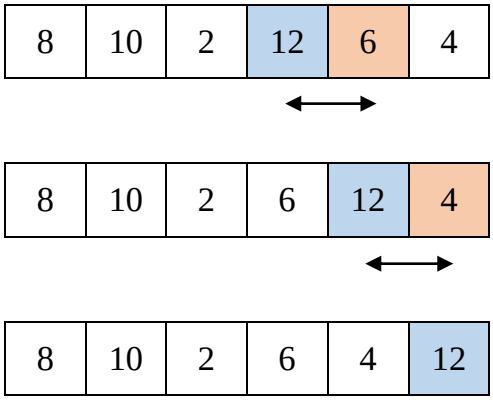
Initially: Ta sẽ duyệt các phần tử theo **Bubble Sort**



Duyệt theo chiều xuôi, ta thấy 12>10 nên ta hoán vị 12 và 10

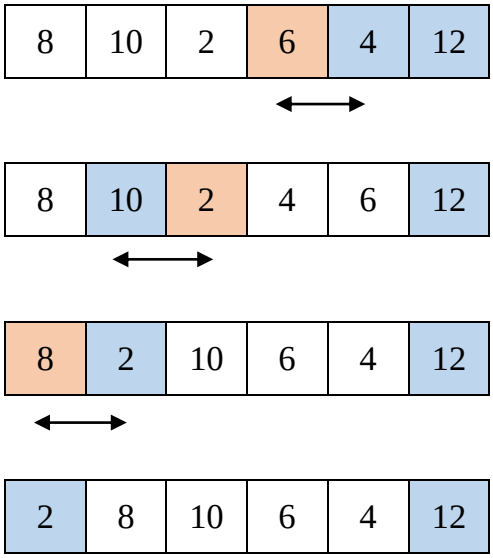
First forward pass: Thực hiện các bước tương tự như trên cho đến cuối mảng





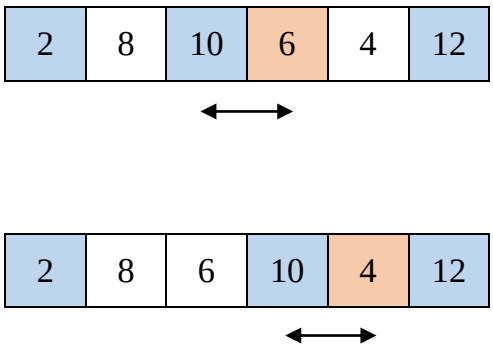
Sau lần duyệt này, phần tử lớn nhất đã đưa về cuối, ta sẽ duyệt ngược lại từ cuối lên đầu để đưa phần tử nhỏ nhất lên đầu mảng, cách duyệt giống **Bubble Sort**

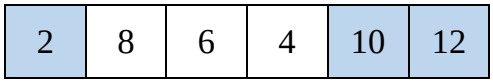
First backward pass:



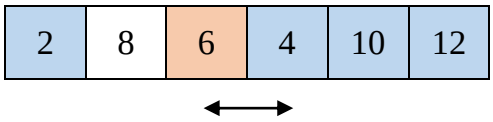
Như vậy, sau hai lần duyệt, ta đã đưa phần tử nhỏ nhất và lớn nhất lần lượt lên đầu mảng và cuối mảng, ta sẽ thực hiện tương tự cho các lần duyệt sau

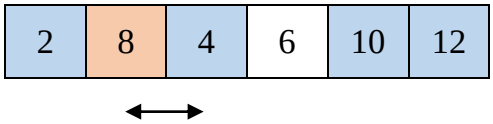
Second forward pass:

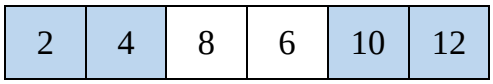




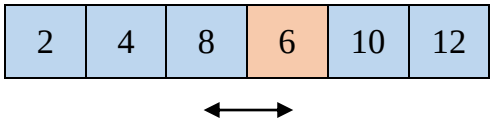
Second backward pass:

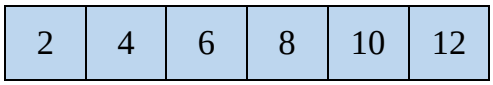






Third fordward pass:





Như vậy, sau khi chạy xong thuật toán, ta đã được mảng sắp xếp tăng dần

3.11.5 Đánh giá độ phức tạp của thời gian, không gian

Độ phức tạp thời gian:

- **Trường hợp tốt nhất (Best case):** $O(n)$, khi mảng được sắp xếp tăng dần, thuật toán sẽ dừng sau khi chạy vòng lặp đầu tiên .
- **Trường hợp trung bình (Average case):** $O(n^2)$, khi mảng ngẫu nhiên, giống Bubble Sort nhưng hiệu quả hơn vì duyệt 2 chiều.
- **Trường hợp tệ nhất (Worst case):** $O(n^2)$, tương đương với Bubble Sort khi mảng cho được sắp xếp ngược lại so với yêu cầu.

Độ phức tạp không gian: `O(1)`, vì thuật toán chỉ yêu cầu `O(1)` không gian cho mảng ban đầu.

4. Kết quả thực nghiệm và nhận xét

Đối với phần này, ta sẽ thống kê số liệu và nhận xét dựa trên các yếu tố sau:

- **Các thứ tự mảng sử dụng:** Mảng được sắp xếp (sorted), mảng được sắp xếp gần hoàn chỉnh (nearly sorted), mảng sắp xếp ngược (reversed) và mảng ngẫu nhiên (randomized).
- **Kích thước của từng mảng:** 10000, 30000, 50000, 100000, 300000, 500000.
- **Chỉ số cần thống kê:** Thời gian chạy (running time) và số phép so sánh (comparisons).

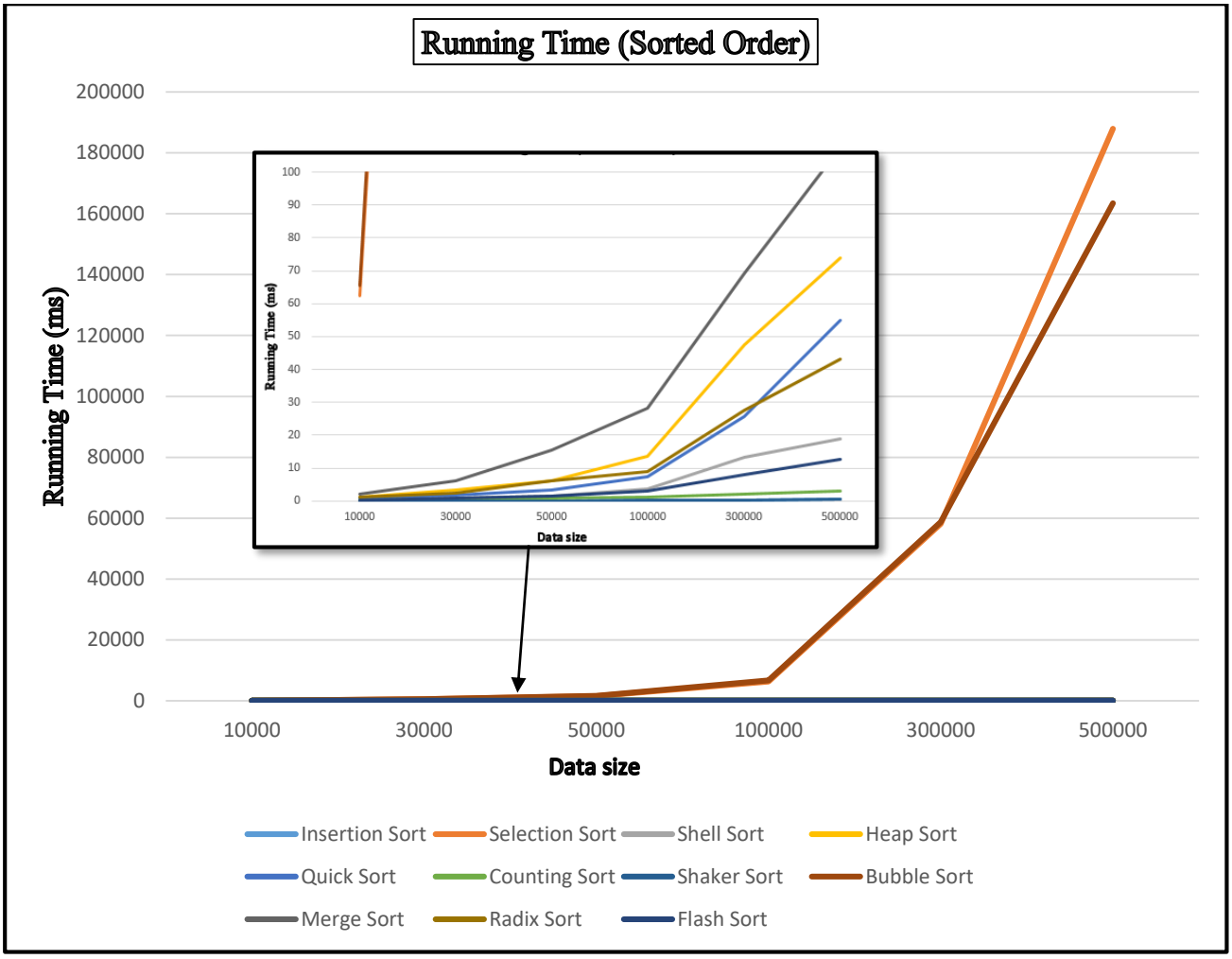
Sau đó, dựa vào bảng số liệu và đồ thị được dựng, ta sẽ đưa ra đánh giá và nhận xét về mức độ phù hợp của từng nhóm thuật toán đối với thứ tự mảng được sử dụng.

4.1 Dữ liệu đã được sắp xếp

Data order: Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Result	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Insertion Sort	0.011	29998	0.032	89998	0.069	50000	0.136	299998	0.305	899998	0.536	1499998
Selection Sort	62.47	100009999	545.1	900029999	1572.41	2500049999	6273.46	10000099999	58110.4	90000299999	187904.29	250000499999
Shell Sort	0.259	360042	0.886	1170050	1.57	2100049	3.61	4500051	13.27	15300061	18.88	25500058
Heap Sort	1.22	668594	3.39	2239075	6.11	3928746	13.7	8360329	47.44	27432145	74.06	47123856
Quick Sort	0.791	138802	1.83	466664	3.32	820914	7.41	1756704	25.88	5782983	55.05	9994919
Counting Sort	0.246	72762	0.533	152765	0.715	232766	1.08	432766	2.22	1232766	3.13	2032766
Shaker Sort	0.014	20002	0.027	60002	0.037	100002	0.101	200002	0.282	600002	0.561	1000002
Bubble Sort	65.37	100009999	572.37	900029999	1620.02	2500049999	6728.16	10000099999	58758.71	90000299999	163482.94	250000499999
Merge Sort	1.97	475242	6.19	1559914	15.46	2722826	28.15	5745658	69.15	18645946	107.58	32017850
Radix Sort	1.13	170074	2.41	510074	6.11	850074	8.85	1700074	27.63	5100074	43.12	8500074
Flash Sort	0.358	125103	0.944	367263	1.55	597048	2.89	1144994	7.87	3434987	12.55	5724979

Bảng 1: Thời gian và tổng số phép so sánh của từng thuật toán với dữ liệu đã được sắp xếp

4.1.1 Thời gian



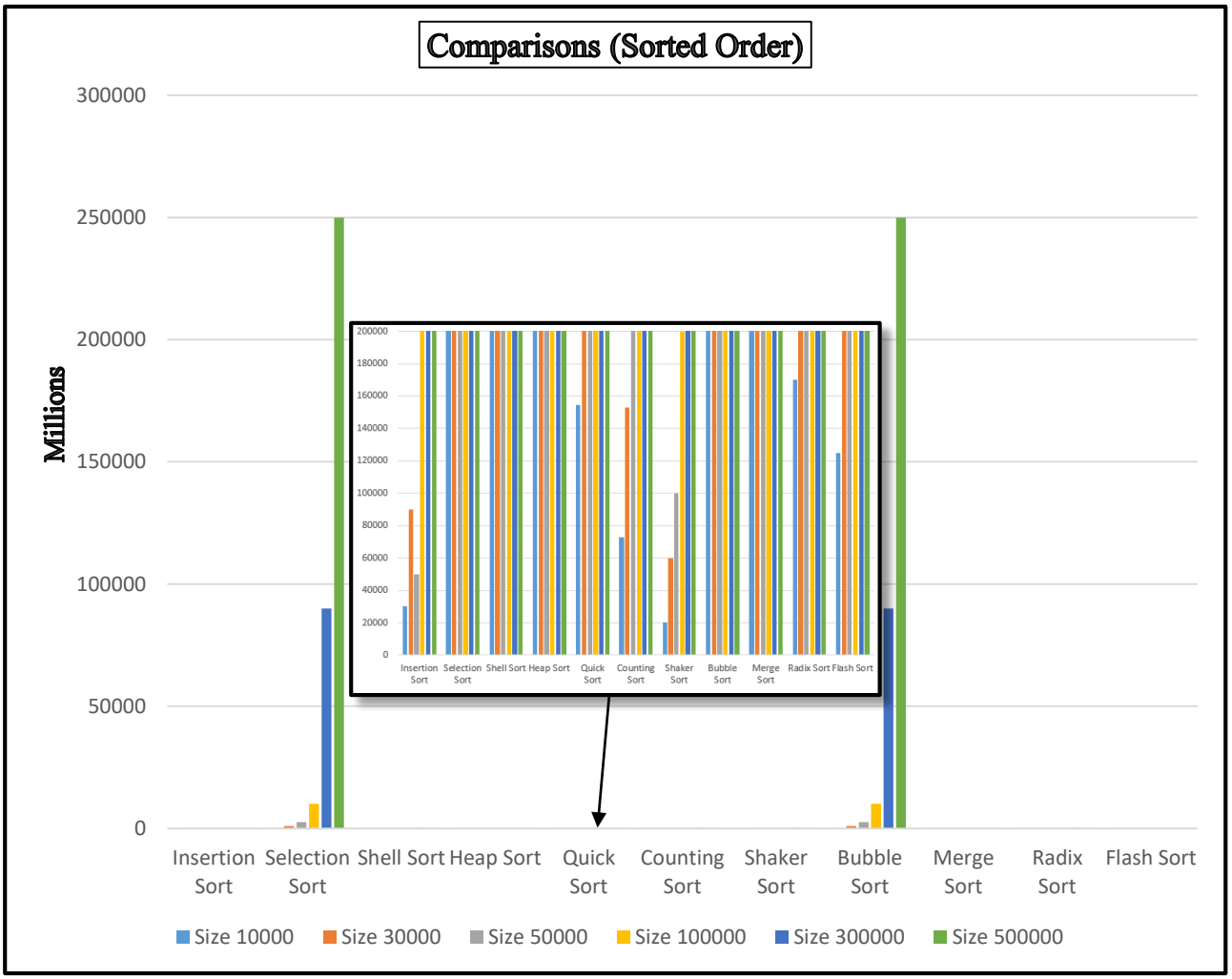
Hình 1: Thời gian chạy từng thuật toán với dữ liệu đã được sắp xếp

Đối với mảng được sắp xếp:

- Thời gian chạy chậm nhất: **Selection Sort.**
- Thời gian chạy nhanh nhất: **Insertion Sort**, **Shaker Sort**.
- Selection Sort, Bubble Sort: Bộ dữ liệu mặc dù được sắp xếp, nhưng chúng luôn phải liên tục kiểm tra các cặp hạng tử có trong mảng để hoán đổi vị trí, như vậy thời gian chạy vẫn tăng theo kích thước theo độ lớn của dữ liệu.
- Insertion Sort, Shaker Sort: Hai thuật toán này thể hiện rất tốt khi dữ liệu đã được sắp xếp, điều này xuất phát từ việc bản thân hai thuật toán này có thể kiểm tra được kiểu dữ liệu đặc trưng này, khi đó số phép so sánh được giảm đi rất nhiều, trong trường hợp này, thời gian thực hiện có thể tiến tới `O(n)` và nằm trong nhóm tốt nhất.

- Heap Sort, Merge Sort, Quick Sort: Đây là nhóm có tốc độ cũng khá tốt với độ phức tạp `O(nlogn)`. Khi bộ dữ liệu càng được mở rộng, tốc độ bản của nhóm này tăng chậm và cho thấy khả năng xử lí dữ liệu lớn vẫn rất tốt.
- Shell Sort: Là thuật toán cải tiến của Insertion Sort có độ phức tạp trung bình `O(nlogn)` nhưng lại không thể kiểm tra kiểu dữ liệu sắp xếp, thành ra tốc độ không được tối ưu như Insertion Sort.
- Flash Sort, Radix Sort, Counting Sort: Nhóm này dựa vào khả năng phân phối của dữ liệu để thực hiện sắp xếp, với mảng đã được sắp xếp, hiệu năng cũng được cải thiện và cho ra tốc độ cũng rất nhanh.

4.1.2 Phép so sánh



Hình 2: Số phép so sánh của từng thuật toán với dữ liệu đã được sắp xếp

Đối với mảng được sắp xếp:

- Phép so sánh nhiều nhất: **Bubble Sort, Selection Sort.**
- Phép so sánh ít nhất: Insertion Sort.
- Bubble Sort, Selection Sort: Hai thuật toán này luôn phải so các cặp phần tử trong mảng với nhau, nên mặc dù dữ liệu đã được sắp xếp, chúng luôn kiểm tra tất cả các cặp phần tử.
- Insertion Sort: Vì có khả năng nhận biết dữ liệu đã được sắp xếp sẵn, thuật toán chỉ cần duyệt qua một lần các phần tử trong mảng nên số phép giảm đi rất nhiều.
- Quick Sort, Heap Sort: Vẫn phụ thuộc vào kích thước mảng để duyệt và so sánh các phần từ ở hai nửa mảng khi chọn pivot ở giữa. Ở phía Heap Sort do xài cấu trúc dữ liệu Heap nên phải cần nhiều phép so sánh để tạo ra cấu trúc dữ liệu này.
- Merge Sort: Vì mỗi lần hợp nhất hai mảng, đều phải duyệt các phần tử trong các mảng con sắp xếp, nên số phép so sánh không thay đổi dù mảng đã được sắp xếp so với các kiểu mảng khác.
- Counting sort, Radix sort, Flash sort: Các thuật toán này sắp xếp đều không phụ thuộc vào việc so sánh giữa giá trị của các phần tử nên có số phép so sánh nhiều trung bình và không tăng nhanh khi kích thước mảng tăng lên vì nó không dựa vào việc so sánh phần tử.

4.1.3 Nhận xét chung

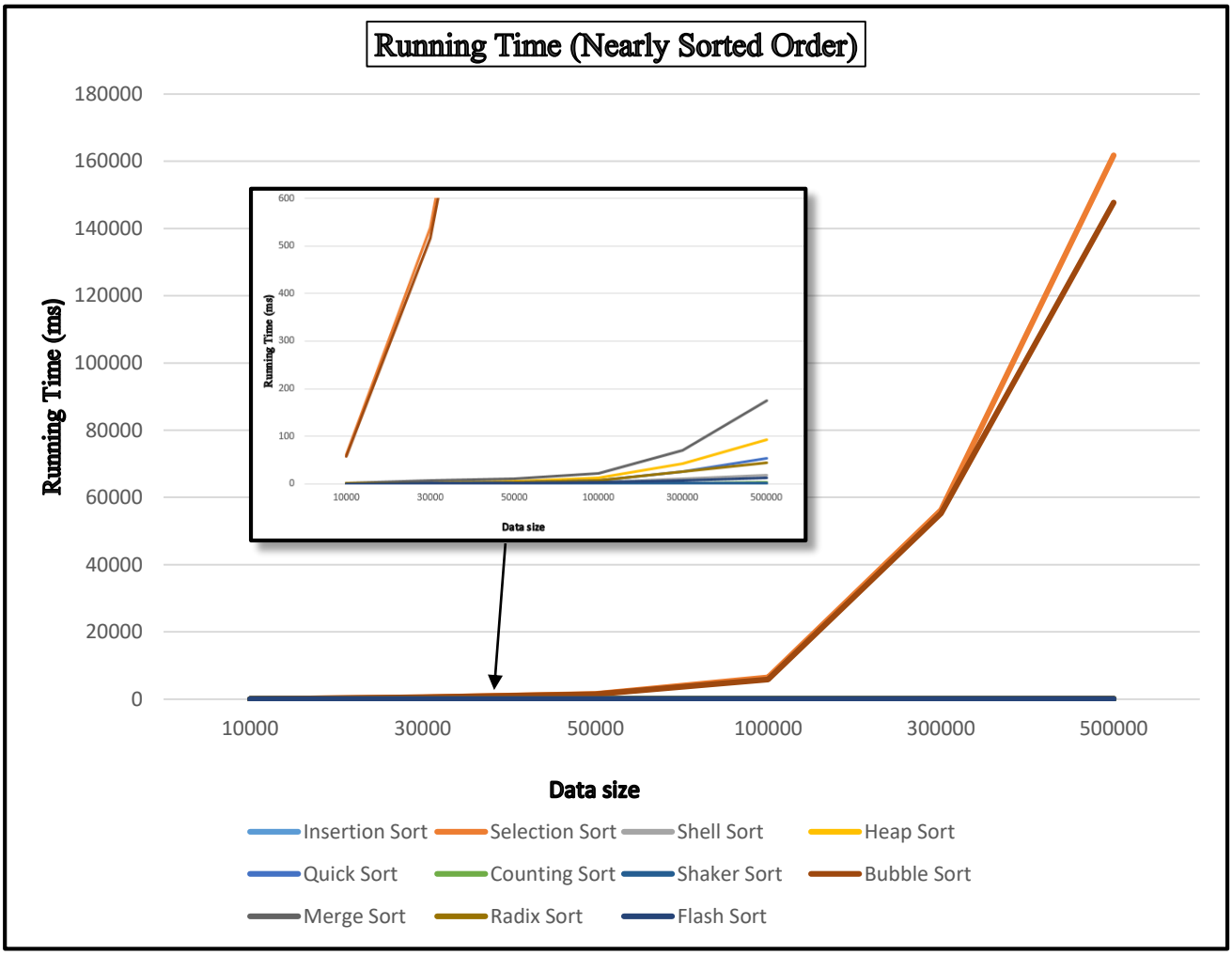
- Đối với các thuật toán Bubble Sort, Selection Sort với độ phức tạp là $O(n^2)$ thì cả thời gian và số phép so sánh cho ra là rất tệ.
- Riêng với Insertion Sort và Shaker Sort lại là những thuật toán hiếm hoi có độ phức tạp trung bình $O(n^2)$ nhưng lại chạy rất nhanh do đặc thù của kiểu dữ liệu đã được sắp xếp.
- Các thuật toán còn lại nhìn chung không bị ảnh hưởng nhiều bởi kiểu dữ liệu đã sắp xếp mà phụ thuộc nhiều vào các yếu tố khác như kích thước mảng, giá trị trong mảng,…

4.2 Dữ liệu gần được sắp xếp hoàn chỉnh

Data order: Nearly Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Result	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp	Time	Comp
Insertion Sort	0.068	132462	0.202	554106	0.213	527302	0.311	607936	0.588	742158	0.962	952834
Selection Sort	61	100009999	538	900029999	1497	2500049999	6428	10000099999	56256	90000299999	161813	250000499999
Shell Sort	0.284	405952	1	1311014	2	2294103	3	4661311	11	15372029	19	25555818
Heap Sort	1	668376	3	2239040	6	3928499	13	8360500	43	27431467	93	47124143
Quick Sort	0.45	154559	1	528007	3	941669	7	2064272	26	7131487	54	12627157
Counting Sort	0.224	72762	0.412	152765	0.642	232766	1.04	432766	2.01	1232766	2.91	2032766
Shaker Sort	0.109	154358	0.435	621354	0.444	637791	0.554	759008	0.622	1087486	0.777	1527982
Bubble Sort	57.98	100009999	516.82	900029999	1431.19	2500049999	5873.57	10000099999	55229.2	90000299999	147741.66	250000499999
Merge Sort	1.92	505226	6.85	1642332	11.02	2828957	22.64	5859386	71.22	18760777	174.94	32118609
Radix Sort	0.822	170074	2.32	510074	4.28	850074	7.51	1700074	25.45	5100074	43.81	8500074
Flash Sort	0.344	125093	0.875	367251	1.5	597036	2.83	1144994	7.65	3434987	12.83	5724979

Bảng 2: Thời gian và tổng số phép so sánh của từng thuật toán với dữ liệu gần được sắp xếp

4.2.1 Thời gian

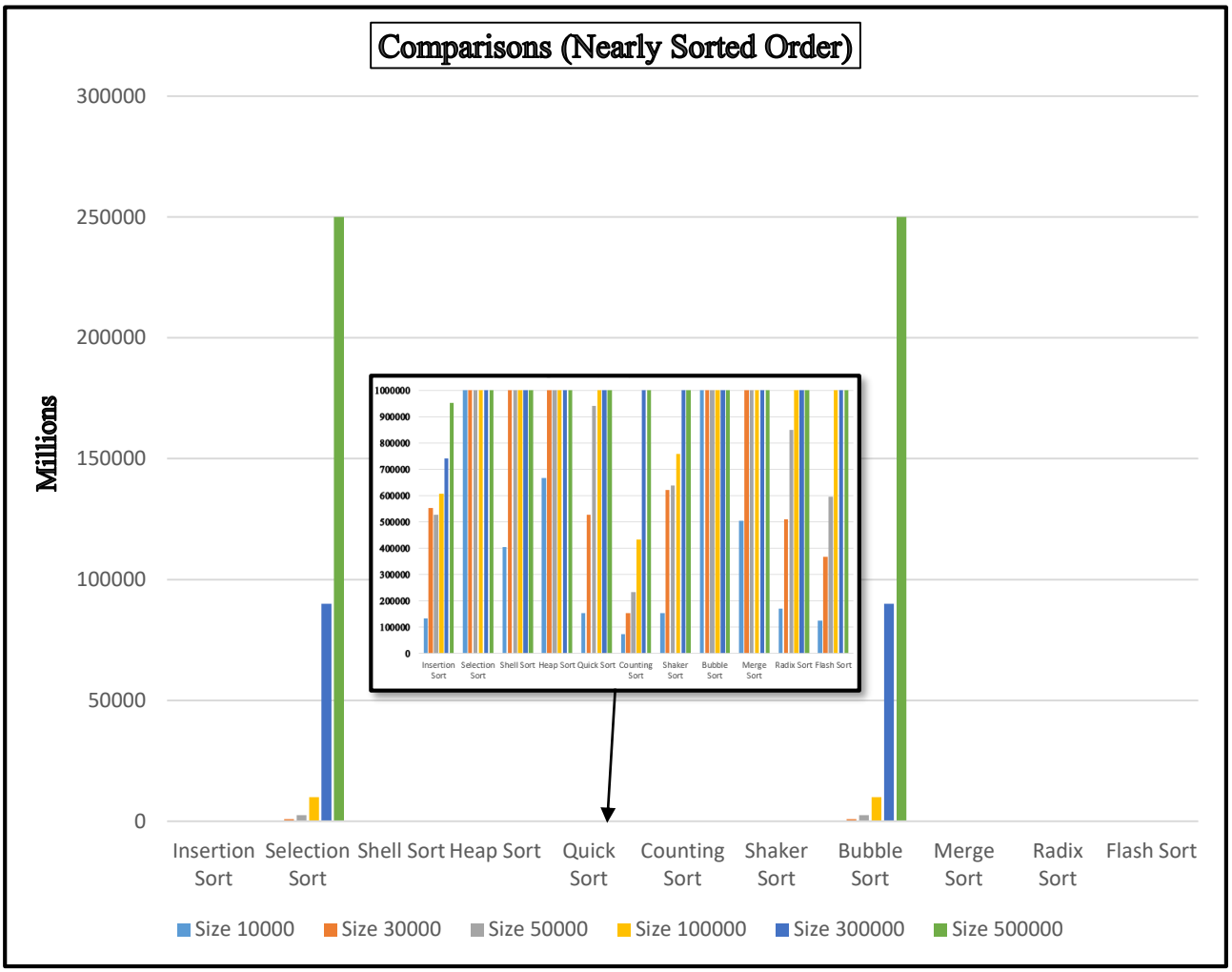


Hình 3: Thời gian chạy từng thuật toán với dữ liệu gần được sắp xếp

Đối với mảng gần như được sắp xếp:

- Thời gian chạy chậm nhất: **Selection Sort.**
- Thời gian chạy nhanh nhất: **Insertion Sort, Shaker Sort.**
- Selection Sort, Bubble Sort: Dù kiểu dữ liệu như nào đi chăng nữa, hai thuật toán này luôn kiểm tra tối đa các cặp phần tử trong mảng, nên tốc độ và phép so sánh luôn lớn.
- Insertion Sort, Shaker Sort: Hai thuật toán này thể hiện rất tốt khi dữ liệu gần đã được sắp xếp, tương tự như như kiểu dữ liệu đã được sắp xếp, chúng chỉ thêm vài thao tác hoán đổi các vị trị nhưng do rất ít phần tử sai vị trí nên thời gian chạy tương đối nhanh.
- Các thuật toán còn lại nhìn chung giống với kết luận của mảng đã được sắp xếp, có lẽ sự có vài thuật cũng có vài sự chênh lệch nhưng nhìn chung trong nhóm này thì chúng không bị ảnh hưởng nhiều.

4.2.2 Phép so sánh



Hình 4: Số phép so sánh của từng thuật toán với dữ liệu gần được sắp xếp

Đối với mảng gần như được sắp xếp:

- Phép so sánh nhiều nhất: **Bubble Sort, Selection Sort.**
- Phép so sánh ít nhất: **Insertion Sort**.
- Bubble Sort, Selection Sort: Hai thuật toán này luôn phải so các cặp phần tử trong mảng với nhau, nên mặc dù dữ liệu đã được sắp xếp, chúng luôn kiểm tra tất cả các cặp phần tử.
- Insertion Sort: Vì có khả năng nhận biết dữ liệu đã được sắp xếp sẵn, thuật toán chỉ cần duyệt qua một lần, nhưng lần này thì phải duyệt vài vòng lặp While để đưa số ít phần tử sai chỗ về đúng vị trí của mình, so sánh với Insertion Sort của mảng đã được

sắp xếp thì nó tăng khá mạnh nhưng so với các thuật toán khác trong mục này thì phép so sánh vẫn tương đối bé hơn.

- Các thuật toán còn lại nhìn chung giống với kết luận của mảng đã được sắp xếp, có lẽ sự có vài thuật cũng có vài sự chênh lệch nhưng nhìn chung trong nhóm này thì chúng không bị ảnh hưởng nhiều.

4.2.3 Nhận xét chung

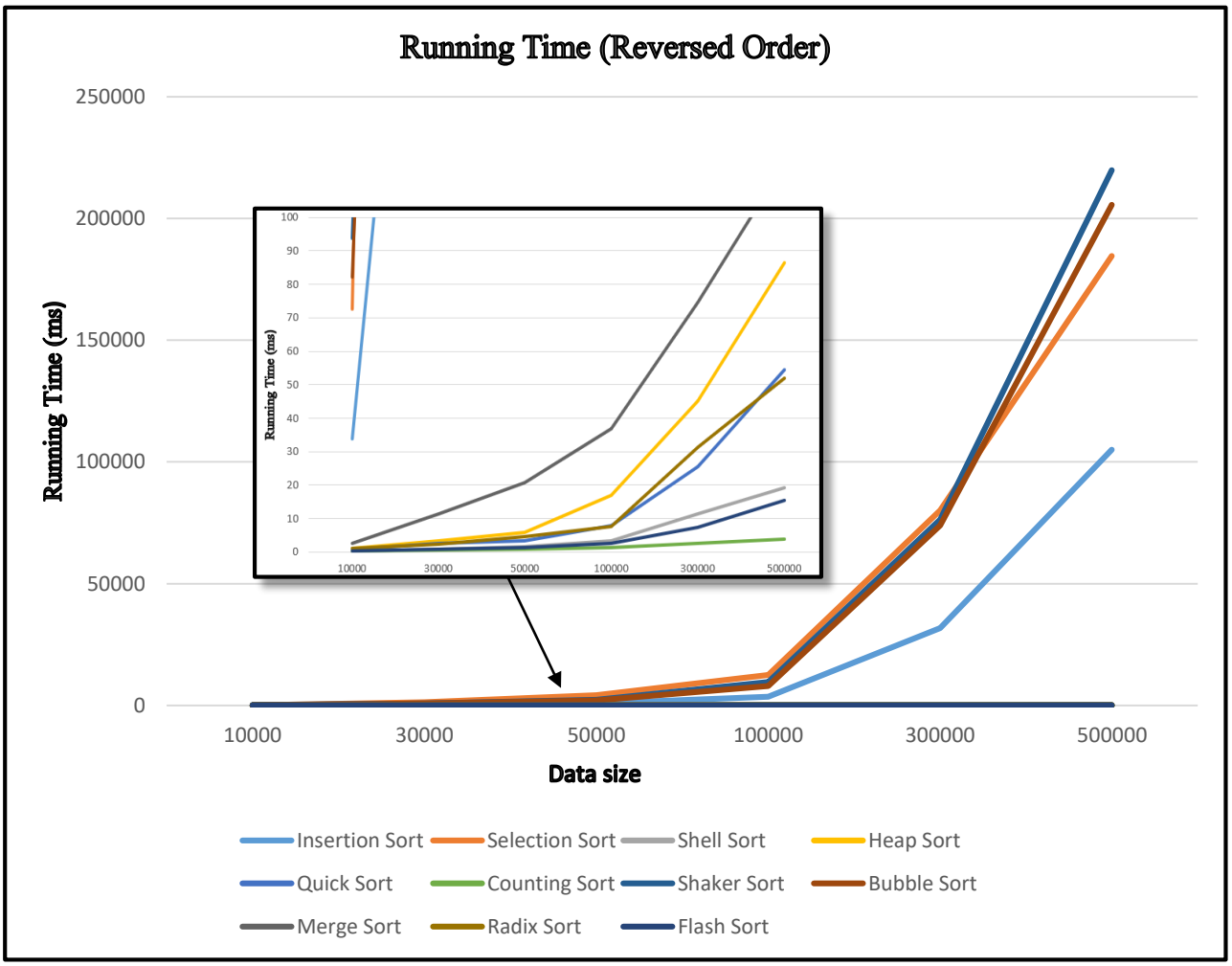
- Nhìn chung đối với mảng gần như sắp xếp, các sự thay đổi về thời gian và số phép so sánh giữa các thuật toán với nhau nói chung và bản thân thuật toán nói riêng không có sự khác biệt quá lớn hay điểm gì nổi bật.
- Thuật toán chậm nhất và có phần lép vế gồm có Bubble Sort và Selection Sort.
- Mặt khác Insertion Sort vẫn giữ được khả năng thể hiện tốc độ vượt trội mặc dù là thuật toán cố độ phức tạp $O(n^2)$.
- Các thuật toán còn lại chạy ổn định và hầu như không có sự thay đổi lớn khi mảng được sắp xếp gần như tăng dần.

4.3 Dữ liệu đã sắp xếp, nhưng theo thứ tự ngược

Data order: Reversed												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Result	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**
Insertion Sort	33.74	100005578	298.86	899992181	825.56	2499949714	3496.4	9999724949	31764	89997288555	105024	249992402705
Selection Sort	72.63	100009999	1322.93	900029999	4184.8	2500049999	12447	10000099999	80101	90000299999	184598	250000499999
Shell Sort	0.26	473909	0.86	1547727	1.65	2803232	3.42	5946672	11.33	19660161	19.17	32815382
Heap Sort	0.987	606703	3.28	2062993	5.89	3612760	16.96	7719016	45.21	25561525	86.38	44450752
Quick Sort	0.884	169561	2.61	572926	3.44	1016618	7.96	2214222	25.43	7581384	54.36	13377023
Counting Sort	0.313	72762	0.59	152765	0.922	232766	1.47	432766	2.63	1232766	3.8	2032766
Shaker Sort	93.88	100004996	822.03	900015001	2302.1	2500025001	9563.9	10000049996	76355	90000149996	219895	250000249975
Bubble Sort	82.12	100009999	741	900029999	1994.8	2500049999	7991.3	10000099999	73946	90000299999	205616	250000499999
Merge Sort	2.56	477554	11.36	1581043	20.81	2760619	36.88	5853651	74.51	19078229	116.21	32852907
Radix Sort	0.979	170074	2.24	510074	4.7	850074	7.61	1700074	31.27	5100074	52.05	8500074
Flash Sort	0.279	118355	0.786	347235	1.34	567393	2.52	1094995	7.32	3284997	15.42	5474994

Bảng 3: Thời gian và tổng số phép so sánh của từng thuật toán với dữ liệu bị đảo ngược

4.3.1 Thời gian



Hình 5: Thời gian chạy từng thuật toán với dữ liệu bị đảo ngược

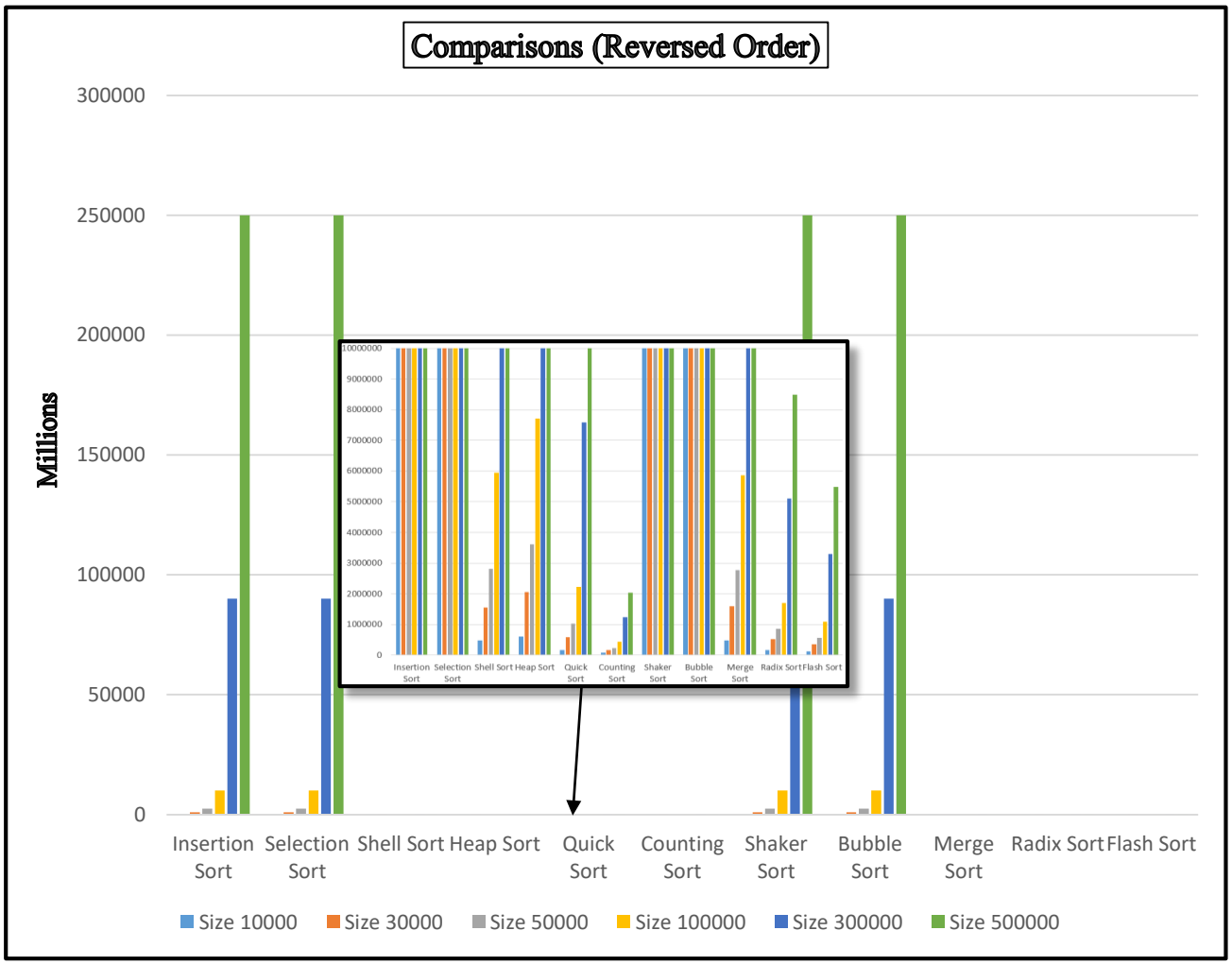
Đối với mảng được sắp xếp đảo ngược:

- Thời gian chạy chậm nhất: **Shaker Sort**
- Thời gian chạy nhanh nhất: **Counting Sort**
- Các thuật toán thuộc nhóm độ phức tạp $O(n^2)$ bao gồm Insertion Sort, Selection Sort, Shaker Sort và Bubble Sort có thời gian chạy tăng rất nhanh khi bộ dữ liệu ngày càng lớn. Lí do là vì các thuật toán này phải duyệt qua danh sách nhiều lần, thực hiện nhiều phép so sánh và hoán đổi vị trí các cặp phần tử.
- Các thuật toán thuộc nhóm độ phức tạp $O(n\log n)$ bao gồm Shell Sort, Heap Sort, Quick Sort và Merge Sort có thời gian chạy tăng ổn định và hoạt động hiệu quả hơn đối với bộ dữ liệu lớn. Lí do là vì các thuật toán này sử dụng phương pháp chia để trị (Merge

Sort, Quick Sort) hoặc sử dụng cấu trúc heap (Heap Sort). Điều này sẽ làm giảm đáng kể số phép so sánh cần thực hiện khi sắp xếp các phần tử.

- Các thuật toán thuộc nhóm độ phức tạp gần với `O(n)` bao gồm Counting Sort, Radix Sort và Flash Sort có thời gian chạy nhanh hơn rất nhiều so với các thuật toán trên. Sự khác biệt của nhóm thuật toán này nằm ở chỗ chúng không thực hiện so sánh riêng biệt từng phần tử mà dựa trên tính chất đặt biệt của dữ liệu (Counting Sort, Radix Sort), hoặc kết hợp phân chia dữ liệu theo phân phối (Flash Sort).

4.3.2 Phép so sánh



Hình 6: Số phép so sánh của từng thuật toán với dữ liệu bị đảo ngược

Đối với mảng được sắp xếp đảo ngược:

- Phép so sánh nhiều nhất: **Bubble Sort, Selection Sort**
- Phép so sánh ít nhất: **Counting Sort**

- Các thuật toán thuộc nhóm độ phức tạp $O(n^2)$ bao gồm Insertion Sort, Selection Sort, Shaker Sort và Bubble Sort có số phép so sánh tăng rất nhanh khi bộ dữ liệu ngày càng lớn. Đặc biệt là đối với mảng bị đảo ngược, các thuật toán phải thực hiện số phép so sánh tối đa để kiểm tra toàn bộ các phần từ và sắp xếp lại chúng.
- Các thuật toán thuộc nhóm độ phức tạp $O(nlogn)$ bao gồm Shell Sort, Heap Sort, Quick Sort và Merge Sort có số phép so sánh tăng ổn định hơn. Tuy nhiên, Quick Sort có thể rơi vào tình huống xấu nhất nếu chọn pivot không phù hợp (luôn là đầu hoặc cuối).
- Các thuật toán thuộc nhóm độ phức tạp gần với $O(n)$ bao gồm Counting Sort, Radix Sort và Flash Sort có số phép so sánh ít hơn rất nhiều, do chúng không so sánh trực tiếp giữa tất cả các phần tử mà dựa vào các chữ số hay phân bố dữ liệu.

4.3.3 Nhận xét chung

Đối với các thuật toán thuộc nhóm $O(n^2)$, chúng có thể đơn giản và dễ triển khai nhưng lại có hiệu năng kém đối với bộ dữ liệu lớn, đặc biệt là mảng bị đảo ngược.

Đối với các thuật toán thuộc nhóm $O(nlogn)$, chúng thích hợp hơn với bộ dữ liệu lớn và có hiệu năng ổn định hơn.

Đối với các thuật toán thuộc nhóm gần $O(n)$, chúng không bị ảnh hưởng bởi thứ tự đầu vào và có hiệu năng tối ưu khi đạt điều kiện thích hợp. Tuy nhiên những thuật toán này có thể yêu cầu thêm mảng phụ.

4.4 Dữ liệu có thứ tự ngẫu nhiên

Data order: Randomized												
Data size	**10000**		**30000**		**50000**		**100000**		**300000**		**500000**	
Result	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**	**Time**	**Comp**
Insertion Sort	17	49615543	150	448444875	412	1250335118	1801	5015375207	17869	45063397031	43146	125243807025
Selection Sort	58	100009999	553	900029999	1503	2500049999	6465	10000099999	54908	90000299999	167435	250000499999
Shell Sort	1	653868	3	2350796	6	4432875	13	9984676	44	33450732	76	62381907
Heap Sort	1	637854	4	2150474	8	3771309	20	8044501	60	26488338	106	45968177
Quick Sort	1	279374	2	953633	4	1626538	9	3452635	34	11376614	64	19932411
Counting Sort	0.242	72762	0.429	152765	1	232766	1	432766	2	1232766	3	2032766
Shaker Sort	111	66073111	1104	596857612	3198	1665992407	12545	6687208261	104410	60042674094	298017	167054464210
Bubble Sort	142	100009999	1519	900029999	4392	2500049999	18239	10000099999	173319	90000299999	456760	250000499999

Merge Sort	2	583677	8	1937608	13	3383040	27	7165959	83	23381800	139	40382367
Radix Sort	1	170074	2	510074	5	850074	8	1700074	26	5100074	44	8500074
Flash Sort	0.417	120897	1	351759	2	561393	5	1140437	22	3210488	56	5506108

Bảng 4: Thời gian và tổng số phép so sánh của từng thuật toán với dữ liệu ngẫu nhiên

4.4.1 Thời gian

Hình 7: Thời gian chạy của từng thuật toán với dữ liệu ngẫu nhiên

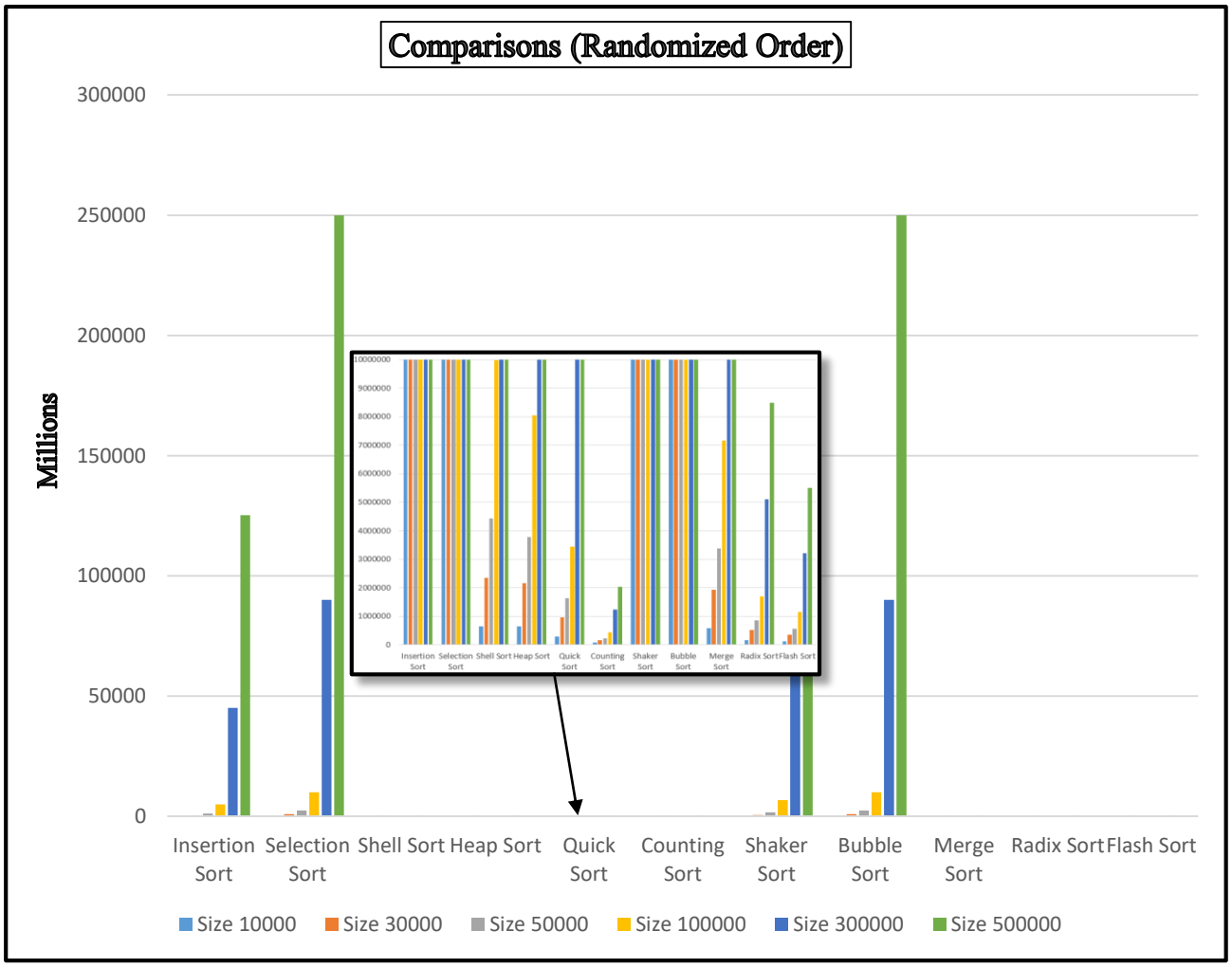
Đối với mảng có thứ tự ngẫu nhiên:

- Thời gian chạy chậm nhất: **Bubble Sort**
- Thời gian chạy nhanh nhất: **Counting Sort**
- Các thuật toán thuộc nhóm độ phức tạp $O(n^2)$ bao gồm Insertion Sort, Selection Sort, Shaker Sort và Bubble Sort có thời gian chạy tăng rất nhanh khi bộ dữ liệu ngày càng

lớn. Lí do là vì các thuật toán này phải duyệt qua danh sách nhiều lần nên thời gian chạy không thay đổi đáng kể đối với mảng ngẫu nhiên.

- Các thuật toán thuộc nhóm độ phức tạp `O(nlogn)` bao gồm Shell Sort, Heap Sort, Quick Sort và Merge Sort có thời gian chạy tăng ổn định và hoạt động hiệu quả hơn đối với bộ dữ liệu lớn. Lí do là vì các thuật toán này sử dụng phương pháp chia để trị (Merge Sort, Quick Sort) hoặc sử dụng cấu trúc heap (Heap Sort).
- Các thuật toán thuộc nhóm độ phức tạp gần với `O(n)` bao gồm Counting Sort, Radix Sort và Flash Sort có thời gian chạy nhanh hơn rất nhiều so với các thuật toán trên. Tuy nhiên, chúng chỉ hiệu quả khi dữ liệu đầu vào phù hợp (Counting Sort và Radix Sort yêu cầu số nguyên) hoặc phân phối đồng đều (Flash Sort). Nếu mảng ngẫu nhiên bị phân bố lệch thì hiệu năng có thể giảm.

4.4.2 Phép so sánh



Hình 8: Số phép so sánh của từng thuật toán với dữ liệu ngẫu nhiên

Đối với mảng được sắp xếp đảo ngược:

- Phép so sánh nhiều nhất: **Bubble Sort, Selection Sort**
- Phép so sánh ít nhất: **Counting Sort**
- Các thuật toán thuộc nhóm độ phức tạp $O(n^2)$ bao gồm Insertion Sort, Selection Sort, Shaker Sort và Bubble Sort có số phép so sánh tăng rất nhanh khi bộ dữ liệu ngày càng lớn.
- Các thuật toán thuộc nhóm độ phức tạp $O(nlogn)$ bao gồm Shell Sort, Heap Sort, Quick Sort và Merge Sort có số phép so sánh tăng ổn định hơn. Merge Sort và Quick Sort duy trì số phép so sánh ổn định, còn Quick Sort thì còn phụ thuộc vào vị trí của pivot.
- Các thuật toán thuộc nhóm độ phức tạp gần với $O(n)$ bao gồm Counting Sort, Radix Sort và Flash Sort có số phép so sánh ít hơn rất nhiều, do chúng không so sánh trực tiếp giữa tất cả các phần tử mà dựa vào các điều kiện đặc thù.

4.1.4 Nhận xét chung

Đối với các thuật toán thuộc nhóm $O(n^2)$, chúng có hiệu năng kém và không phù hợp với mảng ngẫu nhiên có dữ liệu lớn.

Đối với các thuật toán thuộc nhóm $O(nlogn)$, chúng có hiệu năng ổn định và rất hiệu quả cho bộ dữ liệu theo thứ tự này.

Đối với các thuật toán thuộc nhóm gần $O(n)$, chúng có hiệu năng tối ưu nếu bộ dữ liệu phải thoả những điều kiện đặc thù.

4.5 Nhận xét tổng thể

- Thuật toán kém hiệu quả nhất: Bubble Sort, Selection Sort
  - Hai thuật toán này đều cho ra thời gian và số phép so sánh rất lớn mà không phụ thuộc vào kiểu dữ liệu đã được sắp xếp hay chưa.
- Thuật toán hiệu quả tốt nhất mọi hay một số trường hợp :
  - Insertion Sort, Shaker Sort hiệu quả trong các mảng đã được sắp xếp hoặc gần như đã được sắp xếp. Nhưng đối với mảng ngẫu nhiên hoặc sắp xếp ngược thì chạy chậm.
  - Counting Sort và Flash Sort luôn đạt hiệu suất tốt nhất khi đặt so sánh ở tổng thể mọi kiểu mảng với thời gian chạy nhanh và số phép so sánh ít.

- Các thuật toán còn lại Heap Sort, Merge Sort, QuickSort, Shell Sort nhìn chung chạy khá nhanh ở bất kì kiểu mảng nào.

5. Quản lý thư mục dự án

5.1. Tổ chức mã nguồn

Mã nguồn của chương trình được viết bằng ngôn ngữ **C++** trên hệ điều hành **Windows** và được tổ chức trong **Microsoft Visual Studio Community 17.11.5** với cấu trúc như sau:

`source/`

- `SORT_PROJECT/`
  - `header.h`
  - `utils.h`
  - `sortingAlgorithms.h`
  - `baseFunctions.h`
  - `main.cpp`
- `x64/`
  - `Debug/`
    - …
    - `SORT_PROJECT.exe`
  - `Release/`
    - …
    - `SORT_PROJECT.exe`
- `SORT_PROJECT.sln`

Trong đó:

- **`header.h`:** Tệp chứa các thư viện được sử dụng trong chương trình, bao gồm:
  - `iostream`: Thư viện thực hiện nhập/xuất dữ liệu.
  - `fstream`: Thư viện đọc và ghi dữ liệu từ tệp.
  - `string`: Thư viện cung cấp các chức năng để làm việc với chuỗi.
  - `chrono, ctime`: Các thư viện để đo thời gian.
  - `cstdlib`: Thư viện dùng để tạo số ngẫu nhiên trong chương trình.
- **`utils.h`:** Tệp chứa các hàm đọc, ghi mảng và các khởi tạo các mảng theo các thứ tự `randomized, nearly sorted, sorted và reversed.`

- **`sortingAlgorithms.h`:** Tệp chứa các hàm chạy thuật toán sắp xếp trong đồ án.
- **`baseFunctions.h`:** Tệp chứa các hàm đo thời gian chạy, số phép so sánh và các hàm chạy `command 1,2,3,4,5.`
- **`main.cpp`:** Mã nguồn chính của chương trình.
- **`SORT_PROJECT.exe`:** Tệp thực thi được dùng để chạy với các tham số dòng lệnh từ người dùng.
- **`SORT_PROJECT.sln`:** Tệp chứa các thông tin quan trọng (như cấu hình, liên kết,…) của dự án trong Visual Studio.

5.2. Biên dịch chương trình

Để biên dịch chương trình, thiết bị của người dùng phải thoả các điều kiện sau:

- Hệ điều hành Windows (phiên bản được sử dụng trong đồ án là Windows 10).
- Microsoft Visual Studio (phiên bản được sử dụng trong đồ án là 17.11.5).
- Command Prompt (hoặc terminal khác có hỗ trợ chức năng tương ứng).

Người dùng biên dịch chương trình theo cách bước sau:

- **Bước 1:** Mở tệp `SORT_PROJECT.sln` bằng Visual Studio.
- **Bước 2:** Chọn chế độ Debug hoặc Release (đề xuất chọn Release) và ấn tổ hợp phím Ctrl + Shift + B để build dự án.
- **Bước 3:** Tệp thực thi .exe sẽ nằm ở thư mục tương ứng (`source/x64/Debug/` hoặc `source/x64/Release`).
- **Bước 4:** Mở terminal tại thư mục có chứa tệp .exe và tiến hành các tham số để chạy chương trình.

Kết quả của chương trình sẽ được hiển thị trên terminal mà người dùng chạy các lệnh.

6. Tài liệu tham khảo

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 2, pp. 17 – 18, 2022.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 2, pp. 20 – 21, 2022.

[3] S. Kumar and P. Singla, "Sorting using a combination of Bubble Sort," *Mathematical Sciences and Computing*, vol. 2, pp. 31 – 33, 2019.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 4, pp. 40 – 43, 2022.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 4, pp. 102 – 105, 2022.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 8, pp. 209.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 8, pp. 211 – 213.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 8, pp. 213 – 214.

[9] *"The Flashsort1 Algorithm,"* Dr. Dobb's Journal, vol. 23, no. 2, Feb. 1998. Flash Sort

[10] W3resource, "JavaScript Sorting Algorithm: Flash sort," Dec. 2024.

[11] Codelearn.io, "Flash Sort - The Divine Sorting Algorithm," *codelearn.io*, Mar. 18, 2021.

[12] Faro, S., Marino, F. P., & Scafiti, S. (2020). Fast-Insertion-Sort: A new family of efficient variants of the Insertion-Sort algorithm. *In Proceedings of the SOFSEM 2020 Doctoral Student Research Forum* (Vol. 2568, pp. 37-48). Fast Insertion Sort.

[13] GeeksforGeeks. (2024, August 27). Comb Sort Algorithm.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 26, pp. 775 – 776, 2022.

[15] Lê Minh Hoàng, "Giải Thuật và Lập Trình," Hanoi University of Education, 1999-2002, pp. 89.

[16] codelearn.io. Shell Sort. Retrieved January 6, 2025.

[17] R. Kline. (2019, December 20). "Shell Sort Comparison." *West Chester University of Pennsylvania*.

[18] W. Hibbard, "An empirical study of minimal storage sorting," *Communications of the ACM*, vol. 6, no. 5, pp. 206–213, May 1963.

[19] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998.

[20] R. Sedgewick, "Analysis of Shellsort and related algorithms," *Proceedings of the Fourth Annual European Symposium on Algorithms*, Barcelona, Spain, 1996, pp. 1–11.

[21] V. R. Pratt, "Shellsort and Sorting Networks," *Ph.D. dissertation*, Stanford University, 1971.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: The MIT Press, ch. 6, pp. 170 – 171, 2022.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, và C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009, pp.123-145.

[24] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[25] L. R. Weiss, *Data Structures and Algorithm Analysis in C*, 3rd ed., Boston, MA: Pearson, 2006, pp. 196-200.

[26] L. R. Weiss, *Data Structures and Algorithm Analysis in C*, 3rd ed., Boston, MA: Pearson, 2006, pp. 198-202.

[27] J. A. Storer, *An Introduction to Data Structures with Applications*, 2nd ed., Boston, MA: Addison-Wesley, 1995, pp. 357-360.

[28] R. Sedgewick, *Algorithms*, 3rd ed., Boston, MA: Addison-Wesley, 2002, pp. 112-115.

[29] D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Boston, MA: Addison-Wesley, 1998, pp. 110-115.

[30] M. Mitzenmacher và E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge, U.K.: Cambridge University Press, 2005, pp. 59-65.