THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

BÁO CÁO THỰC HÀNH GIỮA KỲ: CÁC THUẬT TOÁN SẮP XẾP

Tên	Nguyễn Đăng Tiến Thành
MSSV	19120036
Lớp	19CTT1TN
Trường	Đại học Khoa học Tự Nhiên - ĐHQG TP. Hồ Chí Minh
Email	19120036@student.hcmus.edu.vn
Ngày	Ngày 1 tháng 12 năm 2020

Mục lục

1. Các thuật toán sắp xếp đã cài đặt

Ở bài báo cáo này trình bày về 12 thuật toán sắp xếp phổ biến được sử dụng để sắp xếp lại một dãy số A có n phần tử theo thứ tự không giảm. Để tiện trong việc mô tả thuật toán, mảng A được đánh chỉ số từ 1 đến n.

1.1 Selection Sort

1.1.1 Ý tưởng

Thuật toán sắp xếp chọn (selection sort) sẽ thực hiện việc sắp xếp dãy số bằng cách lần lượt duyệt qua từng vị trí i và tìm phần tử có giá trị lớn thứ i trong mảng để đổi chỗ với vị trí i này:

- Ở lượt thứ 1, ta sẽ chọn trong mảng $A[1..n]$ phần tử có giá trị nhỏ nhất và đổi giá trị của nó với $A[1]$, khi đó $A[1]$ sẽ trở thành phần tử có giá trị nhỏ nhất
- Ở lượt 2, ta sẽ chọn trong mảng $A[2..n]$ phần tử có giá trị nhỏ nhất và đổi giá trị của nó với $A[2]$, khi đó $A[2]$ sẽ trở thành phần tử có giá trị nhỏ thứ 2
- ...
- Ở lượt thứ i, ta sẽ chọn trong mảng $A[i..n]$ phần tử có giá trị nhỏ nhất và đổi giá trị của nó với $A[i]$, khi đó $A[i]$ sẽ là phần tử nhỏ thứ i.

Sau khi kết thúc quá trình, tức là lặp đến $i = n$, ta sẽ được mảng A mà ở đó $A[i]$ sẽ là phần tử có giá trị nhỏ thứ i trong mảng, và ta được một dãy đã được sắp xếp không giảm.

1.1.2 Thuật toán

```
SelectionSort(arr, n) {
    for (i = 1; i < n; ++i) {
        minPos = indexOfMinimum(arr, from = i, to = n)
        swap(arr[minPos], arr[i])
    }
}
```

1.1.3 Phân tích

Đối với thuật toán sắp xếp chọn, ở lượt thứ i, để chọn được khóa nhỏ nhất trong mảng $A[i..n]$ ta cần $n - i$ phép so sánh, số lượng phép so sánh phải thực hiện không phụ thuộc vào phân bố dữ liệu ban đầu, từ đó suy ra tổng số phép so sánh phải thực hiện:

$$(n-1)+(n-2)+...+1 = n*(n-1)/2 \approx n^2$$

Vậy thuật toán sắp xếp chọn có độ phức tạp là $O(n^2)$ trong mọi trường hợp
Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.2 Insertion Sort

1.2.1 Ý tưởng

Thuật toán sắp xếp chèn (insertion sort), ý tưởng lần lượt xét từng phần tử $A[i]$ và chèn vào mảng $A[1..i-1]$ ở trước đó trong điều kiện đã được sắp xếp, sao cho sau khi chèn $A[1..i]$ cũng được sắp xếp. Với tư tưởng đó, dễ dàng thấy sau khi thực hiện đến phần tử $A[n]$ ta sẽ được dãy $A[1..n]$ được sắp xếp.

1.2.2 Thuật toán

```
InsertionSort(arr, n) {
    for (i = 2; i <= n; ++i) {
        pos = i
        value = arr[i]
        // find correct position to insert arr[i]
        while (pos > 1) and (value < arr[pos - 1]) {
            arr[pos] = arr[pos - 1] // shift right 1
            pos = pos - 1
        }
        arr[pos] = value // insert arr[i]
    }
}
```

1.2.3 Phân tích

Đối với thuật toán sắp xếp chèn, chi phí thời gian thực hiện phụ thuộc vào phép so sánh để tìm vị trí đúng của mỗi $A[i]$ và phụ thuộc vào dãy khóa ban đầu:

- Trong trường hợp tốt nhất, khi dãy đã được sắp xếp sẵn, mỗi lượt xét $A[i]$ ta chỉ cần 1 phép so sánh để tìm được vị trí chèn $A[i]$ đúng, và do đó tổng cộng ta tốn n phép so sánh. Vậy trong trường hợp tốt nhất, giải thuật có độ phức tạp thời gian là: $O(n)$.
- Trong trường hợp xấu nhất, khi dãy có thứ tự ngược, với mỗi lần xét $A[i]$ ta cần tới $i-1$ phép so sánh để tìm vị trí chèn đúng, và do đó ta cần tổng cộng $1+2+...+(n-1) = n*(n-1)/2 \approx n^2$ phép so sánh. Vậy trong trường hợp xấu nhất, độ phức tạp là: $O(n^2)$
- Trong trường hợp trung bình, độ phức tạp là $O(n^2)$

Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.3 Binary Insertion Sort

1.3.1 Ý tưởng

Dựa trên giải thuật sắp xếp chèn thô sơ, ta nhận thấy ở bước tìm vị trí thích hợp để chèn $A[i]$ vào bằng cách tìm kiếm trong mảng đã được sắp xếp $A[1..i-1]$ ở trước đó, thay vì sử dụng tìm kiếm tuần tự với độ phức tạp $O(n)$, ta có thể sử dụng giải thuật tìm kiếm nhị phân để tìm vị trí đó trong thời gian $O(log(n))$.

1.3.2 Thuật toán

```
BinaryInsertionSort(arr, n) {
    for (i = 2; i <= n; ++i) {
        value = arr[i]
        pos = findPositionWithBinarySearch(arr,1,i-1)
        shiftRightOnePosition(arr, pos, i - 1) //doi
           vi tri a[j] -> a[j + 1] voi j = pos..i - 1
        arr[pos] = value // insert arr[i]
    }
}
```

1.3.3 Phân tích

Nhìn chung thuật toán sắp xếp chèn sử dụng tìm kiếm nhị phân được cải tiến hơn so với sắp xếp chèn, giảm chi phí tìm kiếm vị trí xuống $O(log(n))$, tuy nhiên điều đó không làm giảm đi độ phức tạp của thuật toán bởi thao tác chèn vẫn giữ độ phức tạp:

- Trong trường hợp tốt nhất, khi dãy đã được sắp xếp sẵn, mỗi lượt xét $A[i]$, thực hiện thuật toán tìm kiếm nhị phân vẫn phải tốn $O(log(i))$. Nên độ phức tạp của giải thuật sẽ là $O(nlog(n))$, để tối ưu ta có thể so sánh $A[i]$ với $A[i-1]$ trước để bỏ qua bước tìm kiếm nhị phân, độ phức tạp lúc này sẽ trở về $O(n)$
- Trong trường hợp xấu nhất, khi dãy có thứ tự không tăng, thì với mỗi lần xét $A[i]$ ta chỉ cần $O(log(i))$ phép so sánh để tìm ra vị trí chèn đúng, nhưng vẫn phải tốn $i-1$ lần dịch lùi các phần tử sang trái để tạo chỗ trống chèn $A[i]$. Do đó độ phức tạp trong trường hợp xấu nhất vẫn là $O(n^2)$
- Trong trường hợp trung bình, độ phức tạp là $O(n^2)$

Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.4 Bubble Sort

1.4.1 Ý tưởng

Ý tưởng của thuật toán sắp xếp nổi bọt (bubble sort) như tên gọi của nó, với mỗi lần lặp, phần tử nhỏ nhất sẽ nổi lên đầu dãy, và cuối cùng ta được dãy đã sắp xếp.
Duyệt từ cuối lên đầu dãy, mỗi lần gặp một cặp giá trị liền kề có thứ tự không đúng thì đổi chỗ chúng cho nhau, sau lần duyệt như vậy, giá trị nhỏ nhất sẽ được chuyển lên vị trí đầu dãy. Và giờ bài toán trở thành sắp xếp dãy $A[2..n]$. Cứ tiếp tục như vậy sau khi thực hiện việc trên n lần ta thu được dãy đã sắp xếp

1.4.2 Thuật toán

```
BubbleSort(arr, n) {
    for (i = 1; i < n; ++i)
        for (j = n; j > i; --j)
            if (notInCorrectOrder(a[j - 1], a[j]))
                swap(a[j - 1], a[j])
}
```

1.4.3 Phân tích

Thuật toán sắp xếp nổi bọt, có tất cả n lần lặp thao tác duyệt và lần lượt đưa giá trị nhỏ nhất trong mảng còn lại lên đầu (cụ thể với lần lặp thứ i vì đưa về vị trí i). Với lần lặp thứ i, ta thực hiện $n - i$ thao tác so sánh. Do đó tổng cộng số phép so sánh thực hiện là:

$$(n-1) + (n-2) + ... + 1 + 0 = n * (n-1)/2 \approx n^2$$

Có thể thấy thuật toán không hề phụ thuộc vào phân bố dữ liệu ban đầu, nên trong mọi trường hợp độ phức tạp của giải thuật sắp xếp nổi bọt sẽ là $O(n^2)$
Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.5 Shaker Sort

1.5.1 Ý tưởng

Shaker Sort là một dạng cải tiến dựa trên thuật toán sắp xếp nổi bọt.
Ý tưởng của shaker sort là sau khi đưa phần tử nhỏ nhất lên đầu mảng thì sẽ đưa phần tử lớn nhất về cuối mảng. Như vậy với mỗi lần duyệt, shaker sort sẽ đưa hai phần tử về đúng vị trí của nó trong mảng, như vậy shaker sort chỉ thực hiện $n/2$ lần duyệt, trong khi bubble sort sử dụng đến n lần duyệt. Để cải tiến shaker sort, ta lưu lại vị trí cuối cùng mà thao tác hoán đổi vị trí xảy ra, như vậy sẽ giúp thu hẹp vùng cần sắp xếp xuống tốt hơn. Thuật toán sẽ kết thúc khi vùng cần sắp xếp là rỗng, với cải tiến này shaker sort có khả năng nhận biết dãy đã có thứ tự và sớm kết thúc giải thuật.

1.5.2 Thuật toán

Dưới đây là mã giả của thuật toán Shaker sort đã được tối ưu để nhận biết và thu hẹp vùng cần sắp xếp sau mỗi lượt lặp.

```
ShakerSort(arr, n) {
    left = 1, right = n
    while (left < right) {
        last = 0
        for (i = left; i < right; ++i)
            if (notInCorrectOrder(a[i], a[i + 1]) {
                swap(a[i], a[i + 1])
                last = i
            }
        right = last

        for (i = right; i > left; --i)
            if (notInCorrectOrder(a[i - 1], a[i]) {
                swap(a[i - 1], a[i])
                last = i
            }
        left = last
    }
}
```

1.5.3 Phân tích

Với mỗi lần lặp của shaker sort, ta sẽ thu hẹp vùng cần sắp xếp xuống ít nhất 2 phần tử.
Ví dụ: xét mảng A = [1, 2, 3, 0], shaker sort chỉ cần 1 lần duyệt đưa giá trị nhỏ nhất là 0 ở cuối mảng lên đầu mảng, trong khi bubble sort cần tới 4 lần duyệt.
Shaker sort có khả năng nhận ra mảng đã sắp xếp, nên tốt hơn Bubble sort trong nhiều trường hợp.

- Trong trường hợp tốt nhất, khi dãy đã được sắp xếp sẵn, shaker sort chỉ duyệt qua mảng 1 lần và thu hẹp vùng cần sắp xếp xuống thành rỗng. Do đó độ phức tạp trong trường hợp tốt nhất là $O(n)$.
- Trong trường hợp xấu nhất, khi dãy có thứ tự ngược, mỗi lượt lặp thuật toán chỉ đưa được 2 phần tử về đúng vị trí, nên thuật toán có độ phức tạp thời gian là $O(n^2)$
- Trong trường hợp trung bình, độ phức tạp là $O(n^2)$

Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.6 Shell Sort

1.6.1 Ý tưởng

Nhược điểm của giải thuật sắp xếp chèn thể hiện khi vị trí cần chèn nằm quá xa ở gần đầu mảng. Để xử lý trường hợp đó, phương pháp sắp xếp Shell sort ra đời mang lại hiệu quả cao.
Ý tưởng, giải thuật sẽ tiến hành sắp xếp chèn trên các phần tử có khoảng cách xa nhau, nhằm mục đích giảm số lần dịch chuyển của các giá trị nằm ở xa vị trí đúng, sau đó thu hẹp khoảng cách này lại cho đến khi khoảng cách bằng 1, lúc đó dãy chắc chắn được sắp xếp.

1.6.2 Thuật toán

```
ShellSort(arr, n) {
    gaps = [n, n / 2, n / 4, ..., 1]
    for (h in gaps) {
        insertionSortByGap(arr, n, h)
    }
}
```

Trong đó hàm insertionSortByGap(arr, n, h) sẽ thực hiện việc sắp xếp từng phân đoạn các phần tử cách nhau một khoảng là h theo thứ tự tăng dần

1.6.3 Phân tích

Shell sort hoạt động hiệu quả và dễ cài đặt, nhưng rất khó để đánh giá chặt chẽ độ phức tạp thời gian của giải thuật. Yếu tố quyết định đến độ hiệu quả của Shell sort chính là cách chọn dãy bước nhảy.

- Trong trường hợp tốt nhất, khi bộ dữ liệu đầu vào đã có thứ tự sẵn. Giải thuật có độ phức tạp thời gian là $O(nlog(n))$
- Đối với dãy bước nhảy nguyên thủy [n / 2, n / 4, ..., 1] do D.L.Shell đưa ra sẽ có độ phức tạp tính toán trường hợp xấu nhất là $O(n^2)$
- Nếu chọn dãy $[1, 3, 7, 15, ..., 2^i - 1]$ thì độ phức tạp thời gian trong trường hợp xấu nhất sẽ là $O(n^{3/2})$
- Nếu chọn dãy $[1, 2, 3, 4, ..., 2^i 3^j]$ thì sẽ có độ phức tạp thời gian trong trường hợp xấu nhất là $O(nlog(n)^2)$

Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.7 Heap Sort

1.7.1 Ý tưởng

Trước khi đến với sắp xếp vun đống (heap sort), ta cần tìm hiểu về cấu trúc Heap.
Heap là một cấu trúc dữ liệu dạng cây nhị phân hoàn chỉnh, mà ở đó nút cha có giá trị lớn hơn hoặc bằng các nút con. Cụ thể trên mảng heap[1..n] thì heap[i] sẽ có giá trị không bé hơn heap[2 * i] và heap[2 * i + 1]. Dễ dàng thấy heap[1] sẽ giữ giá trị lớn nhất.
Tư tưởng của sắp xếp vun đống ở đây sẽ là dựng lại mảng A ban đầu thành một Heap, sau đó lần lượt hoán đổi giá trị phần tử ở đầu $A[1]$ (là giá trị lớn nhất) với phần tử cuối cùng, sau đó thực hiện điều chỉnh lại $A[1..n-1]$ để đưa nó trở về dạng Heap. Khi thực hiện đến bước thứ i, ta sẽ có $A[1..i-1]$ sẽ là một Heap và $A[i..n]$ sẽ là mảng được sắp xếp không giảm đúng bao gồm các phần tử lớn nhất trong mảng ban đầu. Thực hiện cho đến khi Heap chỉ còn lại 1 nút.

1.7.2 Thuật toán

```
heapify(arr, i, n) { //heapify arr[i..n]
    while (i not leaf) {
        j = indexOfLargestChild(i)
        if (arr[i] >= arr[j]) break //correct position
            on heap
        swap(arr[i], arr[j]) //push value down
        i = j //go to child
    }
}

HeapSort(arr, n) {
    // make heap from arr
    for (i = n / 2; i >= 1; --i)
        heapify(arr, i, n)
    // sorting
    for (i = n; i >= 1; --i) {
        swap(arr[1], arr[i])
        heapify(arr, 1, i - 1)
    }
}
```

1.7.3 Phân tích

Heap là cây nhị phân hoàn chỉnh nên có độ cao là $[log2(n)] + 1$. Với mỗi thao tác vun đống (heapify) tốn chi phí thời gian là $O(log(n))$. Thực hiện việc xây dựng Heap ta gọi thủ tục heapify $n/2$ lần , lúc thực hiện sắp xếp, ta gọi thủ tục heapify n lần, mỗi lần tốn $O(log(n))$. Nên tổng chi phí thời gian của giải thuật sắp xếp vun đống sẽ là

$O(nlogn)$ trong trường hợp tốt nhất, trung bình và xấu nhất
Đặc biệt nếu mảng A chỉ gồm các phần tử bằng nhau thì sắp xếp vun đống chỉ có độ phức tạp thời gian là $O(n)$ do thao tác heapify sẽ dừng ngay ở nút gốc.
Giải thuật sắp xếp vun đống nêu trên tận dụng mảng A nên không tốn thêm bộ nhớ, vậy độ phức tạp không gian sẽ là: $O(1)$

1.8 Merge Sort

1.8.1 Ý tưởng

Giả sử có hai dãy A và B đã được sắp xếp, ta có thể tạo ra dãy C là hợp của dãy A và B cũng được sắp xếp, bằng cách so sánh giá trị của phần tử ở đầu mỗi dãy A và B, giá trị nào nhỏ hơn thì cho vào C và loại giá trị đó ra khỏi dãy tương ứng. Quá trình tiếp tục cho đến khi cả hai dãy rỗng.
Giải thuật sắp xếp trộn (merge sort) thực hiện phương pháp chia để trị, chia đôi dãy n phần tử ban đầu thành hai dãy liên tiếp có $n/2$ phần tử $A[1..mid]$ và $A[mid + 1..n]$, sắp xếp và sau đó trộn hai dãy này lại bằng giải thuật trộn được mô tả ở trên, như vậy ta thu được dãy n phần tử đã được sắp xếp. Để sắp xếp mảng $n/2$ phần tử ta lại tiếp tục chia thành hai mảng có $n/4$ phần tử, cứ như vậy gọi thủ tục đệ quy cho đến khi mảng chỉ có 1 phần tử thì được coi là đã sắp xếp.

1.8.2 Thuật toán

```
MergeSort(arr, l, r) {
    if (l >= r) return // 1 element
    mid = (l + r) / 2
    MergeSort(arr, l, mid)
    MergeSort(arr, mid + 1, r)
    mergeTwoArray(arr[l..mid], arr[mid + 1..r])
}

MergeSort(arr, n) {
    MergeSort(arr, 1, n)
}
```

1.8.3 Phân tích

Thao tác mergeTwoArray nhận đầu vào là hai mảng có kích thước $n/2$ có độ phúc tạp là $O(n)$
Mỗi lần thủ tục MergeSort(arr, l, r) sẽ gọi đệ quy chia thành hai nhánh, mỗi nhánh sẽ có kích thước giảm đi một nửa, như vậy sẽ có $log(n)$ tầng đệ quy, ở mỗi tầng sẽ tốn tổng chi phí thời gian là $O(n)$ để lần lượt trộn từng dãy con thành đoạn có thứ tự $A[l..r]$ mà hàm MergeSort(arr, l, r) đang quản lý. Như vậy độ phức tạp thời gian của giải thuật sắp xếp trộn sẽ là: $O(nlog(n))$ trong mọi trường hợp, không phụ thuộc vào phân bố dữ liệu đầu vào.
Ở thủ tục mergeTwoArray, ta cần duy trì một mảng phụ để lưu trữ mảng C là kết quả khi trộn A và B để tránh mất mát dữ liệu. Do đó độ phức tạp không gian sẽ là: $O(n)$

1.9 Quick Sort

1.9.1 Ý tưởng

Ý tưởng chủ đạo của phương pháp sắp xếp nhanh (quick sort) là: để sắp xếp một đoạn chỉ có ít hơn 2 phần tử thì không cần phải làm gì cả vì nó được coi là đã sắp xếp. Nếu đoạn có ít nhất 2 phần tử, ta thực hiện chọn một giá trị ngẫu nhiên làm "chốt"(pivot), sau đó phân hoạch đoạn thành hai đoạn con trong đó mọi phần tử có giá trị không lớn hơn pivot sẽ được xếp đứng trước pivot, còn các phần tử có giá trị không nhỏ hơn pivot sẽ được xếp đứng sau pivot. Như vậy ta đã phân hoạch đoạn đang xét thành hai đoạn con mà trong đó, đoạn thứ nhất chỉ chứa các phần tử có giá trị không lớn hơn pivot, đoạn còn lại chỉ chứa các phần tử có giá trị không nhỏ hơn pivot. Nói cách khác, các phần tử ở bên trái pivot đều có giá trị bé hơn hoặc bằng các phần tử nằm bên phải pivot. Tiếp tục sắp xếp hai đoạn mới tạo ra bằng phương pháp tương tự.

1.9.2 Thuật toán

```
quickSort(arr, l, r) {
    if (l >= r) return // < 2 element
    pi = partition(arr, l, r)
    quickSort(arr, l, pi) // arr[l..pi] <= pivot
    quickSort(arr, pi + 1, r) // pivot <= arr[pi+1..r]
}

partition(arr, l, r) {
    pivot = arr[random(l, r)]
    i = l - 1, j = r + 1
    while (i < j) {
        do
            i = i + 1
        while arr[i] < pivot
        do
            j = j - 1
        while arr[j] > pivot
        swap(arr[i], arr[j])
    }
    return j
}

QuickSort(arr, n) {
    quickSort(arr, 1, n)
}
```

1.9.3 Phân tích

Trong giải thuật sắp xếp nhanh, việc chọn pivot là thao tác rất ảnh hưởng đến thời gian thực thi của thuật toán. Bởi tùy vào giá trị của pivot, hai đoạn con được phân hoạch sẽ có kích thước khác nhau.

- Trong trường hợp tốt nhất, là ở mỗi bước chọn pivot để phân đoạn, ta chọn đúng giá trị trung vị của đoạn (giá trị đứng chính giữa khi đoạn được sắp xếp có thứ tự). Lúc đó ở bước phân hoạch thuật toán sẽ chia đoạn hiện tại thành hai đoạn con có kích thước giảm đi một nửa, do đó lời gọi đệ quy sẽ có độ sâu là $log_2(n)$. Vậy trong trường hợp tốt nhất, độ phức tạp thời gian của giải thuật là $O(nlog(n))$
- Trong trường hợp xấu nhất, ở mỗi bước chọn pivot ta chọn đúng phần tử nhỏ nhất hoặc lớn nhất trong đoạn. Khi đó ở bước phân hoạch thuật toán sẽ chia đoạn hiện tại thành hai đoạn con, trong đó 1 đoạn chỉ có 1 phần tử, đoạn còn lại có đến n - 1 phần tử, tức là ở mỗi bước phân hoạch ta chỉ giảm kích thước đoạn đang xét xuống 1 phần tử. Do đó lời gọi đệ quy có độ sâu lên đến n, dễ gây ra hiện tượng tràn stack. Vậy trong trường hợp xấu nhất, thuật toán có độ phức tạp thời gian là $O(n^2)$
- Trong trường hợp trung bình thuật toán có độ phức tạp thời gian là $O(nlogn)$

Do không tốn thêm bộ nhớ, độ phức tạp không gian của giải thuật là: $O(1)$

1.10 Counting Sort

1.10.1 Ý tưởng

Sắp xếp bằng đếm phân phối (distribution counting sort) là một thuật toán sắp xếp không dựa vào phép so sánh, được sử dụng trong trường hợp đặc biệt là giá trị của các phần tử trong mảng cần sắp xếp có giá trị là số nguyên nằm trong đoạn $[0, M]$. Bằng cách đếm số lần xuất hiện của từng giá trị trong mảng và lưu vào mảng $count[0..M]$, trong đó $count[x]$ = có bao nhiêu giá trị x trong mảng ban đầu. Sau đó sắp xếp lại mảng bằng cách đặt $count[0]$ phần tử 0 vào đầu mảng, đặt $count[1]$ phần tử 1 vào ngay sau đó, ..., đặt $count[M]$ phần tử M vào cuối mảng.

1.10.2 Thuật toán

- **Bước 1:** tạo mảng $count[0..M]$, sau đó đếm số lần xuất hiện của từng phần tử trong mang cần sắp xếp A, bằng cách duyệt qua từng phần từ $A[i]$ và tăng $count[A[i]]$ lên 1.

```
count[0..M] = 0
for (int i = 1; i <= n; ++i)
    count[A[i]] += 1
```

- **Bước 2:** Chỉnh sửa lại giá trị mảng $count[]$ bằng cách cộng dồn $count[i] += count[i-1]$, sau bước này, $count[x]$ sẽ trở thành vị trí cuối cùng để đặt giá trị x vào trong mảng sau khi được sắp xếp.

```
for (int i = 2; i <= M; ++i)
    count[i] += count[i - 1]
```

- **Bước 3:** Tạo dãy phụ B và thực hiện sử dụng mảng $count[]$ để sắp xếp lại A lưu vào B, duyệt mọi phần tử $A[i]$ đặt $A[i]$ vào vị trí $count[A[i]]$, sau khi đặt vào đúng vị trí, vị trí cuối cùng có thể đặt giá trị bằng $A[i]$ tiếp theo sẽ tiến về trước, tức giảm $count[A[i]]$ đi 1.

```
for (int i = n; i >= 1; --i) {
    B[count[A[i]]] = A[i]
    count[A[i]] -= 1
}
```

- **Bước 4:** Sao chép B vào A

```
A[1..n] = B[1..n]
```

1.10.3 Phân tích

Counting sort là một phương pháp sắp xếp khá đơn giản, chỉ sử dụng vòng lặp với độ phức tạp tuyến tính.

Độ phức tạp thời gian của giải thuật đếm phân phối là $O(n + M)$ trong đó n là số phần tử trong mảng A và M là giá trị lớn nhất trong mảng A.
Tuy nhiên giải thuật này có nhược điểm là với M quá lớn, cho dù n nhỏ thì cũng sẽ không hoạt động được. Hoặc trong trường hợp mảng A chứa các số âm.
Để giải quyết trong trường hợp các giá trị trong A có số âm hoặc là giá trị nằm trong một đoạn $[L, R]$ kích thước nhỏ. Ta có thể ánh xạ giá trị A về đoạn $[0..R - L]$ bằng cách trừ mỗi $A[i]$ đi một lượng L.
Về không gian, counting sort cần một mảng $count[]$ có M phần tử và một mảng phụ B có n phần tử.
Do đó về **độ phức tạp không gian** của giải thuật sẽ là $O(n + M)$
Giải thuật counting sort nêu trên là một giải thuật sắp xếp ổn định (stability) vì nó bảo toàn thứ tự ban đầu của các phần tử có giá trị bằng nhau.

1.11 Radix Sort

1.11.1 Ý tưởng

Sắp xếp theo cơ số (radix sort) là một giải thuật sắp xếp không dựa trên phép so sánh giá trị của các phần tử. Giải thuật dựa trên việc phân loại và thứ tự phân loại từng kí tự (chữ số).
Radix sort có nhiều phiên bản như Most Significant Digit (MSD) hoặc Least Significant Digit (LSD), và có thể sắp xếp dựa trên một cơ số bất kì như cơ số nhị phân, thập phân, ...
Dưới đây là trình bày về thuật toán sắp xếp theo cơ số thập phân phiên bản LSD, tức là xét từ chữ số thập phân hàng đơn vị đi lên.
Ý tưởng, ta sử dụng một thuật toán sắp xếp ổn định (ví dụ như Counting sort) để sắp xếp mảng theo chữ số hàng đơn vị (chữ số thứ 1), sau đó tiếp tục sắp xếp theo chữ số hàng chục (chữ số thứ 2), do tính ổn định của giải thuật sắp xếp được sử dụng, nên các phần tử có chữ số hàng chục bằng nhau thì phần tử nào có hàng đơn vị nhỏ hơn sẽ xếp trước. Cứ tiếp tục xét đến chữ số cao hơn, dựa vào tính ổn định của thuật toán sắp xếp, dãy thu được sẽ có thứ tự tăng dần về giá trị. Giải thuật kết thúc khi đã sắp xếp xong theo chữ số ở hàng cao nhất (chữ số thứ d)

1.11.2 Thuật toán

```
RadixSort(arr, n) {
    d = getMaximumDigit(arr, n)
    for (k = 1; k <= d; ++k)
        countingSort_by_Kth_digit(arr, n, k)
}
```

Trong đó:
- hàm getMaximumDigit(arr, n) sẽ trả về số lượng chữ số tối đa của các phần tử trong mảng.
- hàm countingSort_by_Kth_digit(arr, n, h) sẽ thực hiện việc sắp xếp ổn định dãy arr theo chữ số thập phân thứ k.

1.11.3 Phân tích

Xét thuật toán sắp xếp ổn định ở đây là counting sort, có độ phức tạp là $O(n + radix)$ với $radix$ là cơ số được sử dụng, cơ số ở đây là hằng số nên có độ phức tạp là $O(n)$.
Thuật toán radix sort gọi hàm sắp xếp trên d lần với d là số lượng chữ số tối đa của các phần tử trong mảng trong cơ số $radix$. Nên giải thuật radix sort trên có độ phức tạp là: $O(n * d)$
Ở phương pháp cài đặt nêu trên, trong lúc sử dụng counting sort như đã mô tả, cần một mảng phụ B có n phần tử và một mảng $count$ chứa $radix$ phần tử, nên độ phức tạp không gian sẽ là $O(n + radix)$
Ta có thể cài đặt radix sort phiên bản MSD sử dụng cơ số nhị phân, với hỗ trợ của các toán tử xử lý bit như shift, and,.. việc cài đặt đơn giản, nhanh hơn, và không cần phải sử dụng mảng phụ (cài đặt tương tự Quick sort).

1.12 Flash Sort

1.12.1 Ý tưởng

Ý tưởng chính của Flash sort chính là dựa trên việc phân bố lại dữ liệu thành các phân lớp nằm kề nhau, trong đó phân lớp nằm ở trước chỉa chứa các phần tử có giá trị nhỏ hơn phân lớp đứng sau, sau đó sử dụng một thuật toán sắp xếp như sắp xếp chèn để sắp xếp lại dãy số.
Flash sort gồm có 3 giai đoạn:

- **Giai đoạn 1:** Phân loại các phần tử:
  Thực hiện việc phân loại dãy A thành m phân lớp, phần tử A_i sẽ thuộc vào phân lớp k_{A_i} được tính toán bởi công thức:

  $$k_{A_i} = 1 + (m-1) * [\frac{A_i - min(A)}{max(A) - min(A)}]$$

  Sau đó sử dụng phương pháp đếm phân phối, để tính được mảng $L[1..m]$ dùng để đánh dấu vị trí cuối cùng của từng phân lớp thuộc mảng. Phân lớp thứ i sẽ nằm từ vị trí $L[i-1]+1..L[i]$.

- **Giai đoạn 2:** Phân hoạch, chu trình hoán vị
  Ta thực hiện việc đưa từng phần tử về đúng phân lớp của nó bằng cách thực hiện chu trình hoán vị:
  1. Lấy phần tử đầu tiên chưa nằm đúng phân lớp: A_i, điều này xảy ra khi $i \leq L[K_{A_i}]$
  2. Sử dụng mảng $L[]$ để xác định vị trí phải đặt A_i, tính K_{A_i} và đặt vào vị trí $L[K_{A_i}]$, vì đã đặt phần tử đó vào đúng phân lớp nên lùi vị trí cuối cùng của phân lớp xuống 1 đơn vị.
  3. Mỗi khi đưa một phần tử về đúng vị trí phân lớp của nó, ta phải lấy phần tử hiện tại đang chiếm chỗ ra và tiếp tục thực hiện việc phân lớp cho nó. Điều này tạo thành một chu trình hoán vị cho đến khi quay trở lại vị trí bắt đầu ở bước 1.

- **Giai đoạn 3:** Sắp xếp
  Sau khi thực hiện giai đoạn 2, ta thu được một mảng đã được phân hoạch theo các phân lớp. Phân lớp đứng trước sẽ chỉ chứa những giá trị bé hơn phân lớp đứng sau.
  Ta thực hiện một giải thuật sắp xếp cơ bản như sắp xếp chèn để sắp xếp lại từng phân lớp.
  Cuối cùng ta thu được mảng đã được sắp xếp.

1.12.2 Thuật toán

```
FlashSort(arr, n) {
    \\parse 1: Elements Classification
    m = int(0.43 * n) // number of block
    L[1..m] = 0
    for (x in arr) L[K(arr[i])] += 1
    for (i = 2 -> n) L[i] += L[i - 1]

    \\parse 2: Elements Permutation
    cnt = 1, i = 1, k = m
    while (cnt <= n) {
        while (i > L[k]) {
            i += 1
            k = K(arr[i])
        }
        x = arr[i]
        while (i <= L[k]) {
            k = K(x)
            y = arr[L[k]] // next
            arr[L[k]] = x //put x to correct block
            x = y
            L[k] -= 1
            cnt += 1 // one more element in correct
               block
        }
    }
    \\parse 3: Elements Ordering
    for (i = 1 -> m)
        insertionSortForBlockIth(arr, i)
}
```

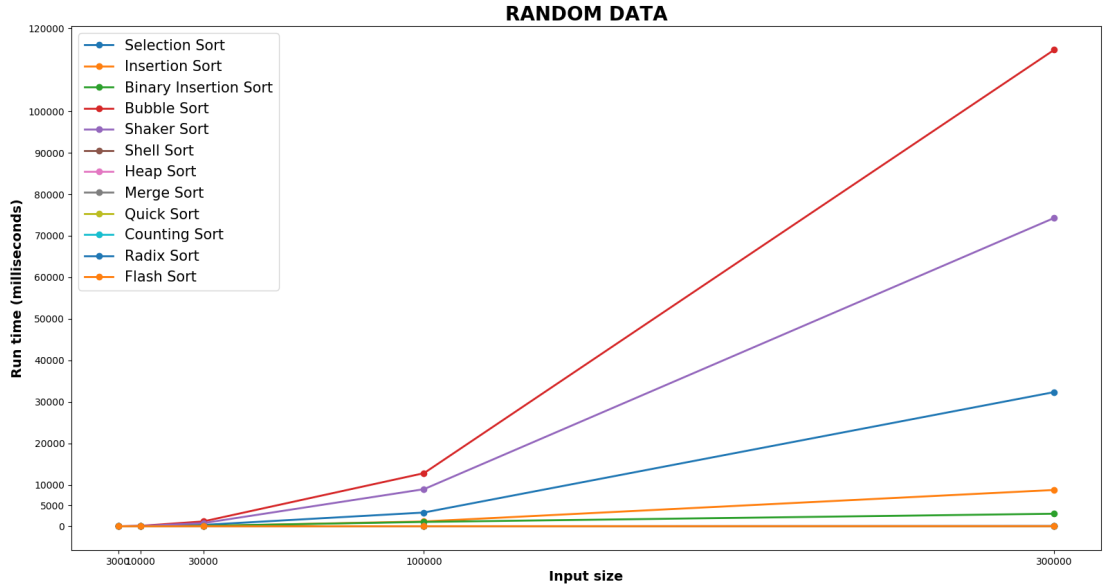
1.12.3 Phân tích

Trong trường hợp dữ liệu được phân bố đều, giải thuật Flash sort hoạt động rất hiệu quả cho độ phức tạp thời gian chỉ trong $O(n)$
Giải thuật Flashsort còn phụ thuộc vào cách chọn số lớp m, trên thực nghiệm người ta nhận thấy rằng với $m \approx 0.42 * n$ sẽ cho ra thời gian thực thi tốt nhất. Thậm chí còn nhanh hơn cả giải thuật sắp xếp nhanh Quick sort.
Độ phức tạp trung bình: $O(n)$
Độ phức tạp không gian: $O(m)$, lưu trữ mảng $L[1..m]$

2. Thực nghiệm

Bộ dữ liệu thực nghiệm được sinh dựa trên mã nguồn $DataGenerator.cpp$, các phần tử trong mảng được sinh ra là các số nguyên nằm trong nửa đoạn $[0, n)$.
Những con số về thời gian thực nghiệm dưới đây được đo thử nghiệm trên một dữ liệu cụ thể, trên một máy tính và trên một công cụ lập trình cụ thể.

2.1 Random order data



Hình 2.1: Biểu đồ thời gian thực thi các thuật toán với bộ dữ liệu ngẫu nhiên

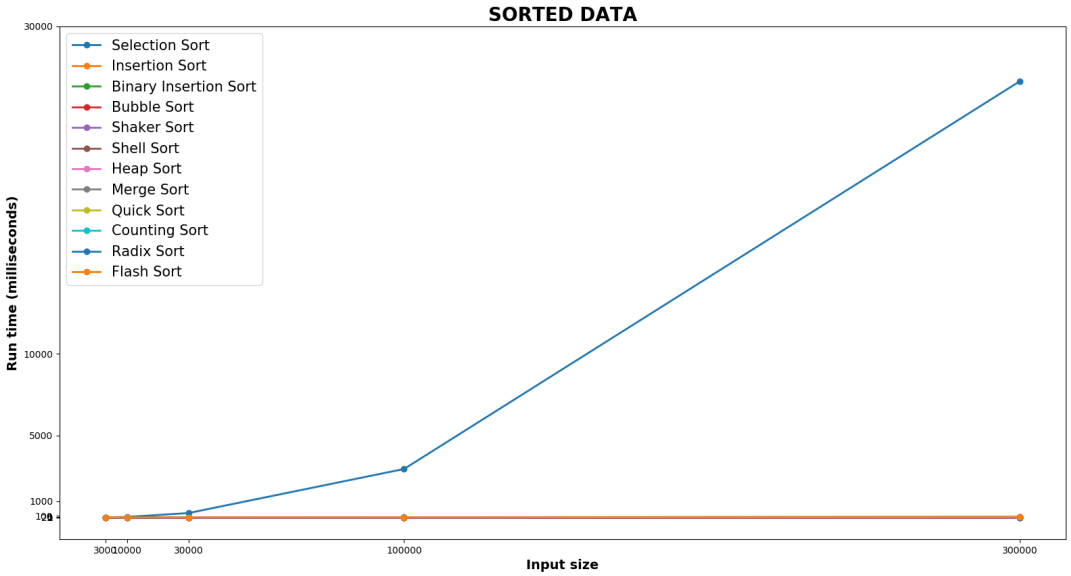
Nhận xét

- Bubble sort tỏ ra là thuật toán có thời gian thực thi chậm nhất trong mọi kích thước dữ liệu đầu vào, với thời gian thực thi lên tới $10^5 ms$ cho bộ dữ liệu 300000 phần tử. Ngay sau đó là thuật toán Shaker sort, và sau đó là Selection sort. Lý do bởi đây là những giải thuật thuộc nhóm có độ phức tạp thời gian $O(n^2)$, với kích thước dữ liệu đầu vào càng lớn thì thời gian thực thi tăng càng nhanh theo đa thức bậc 2.

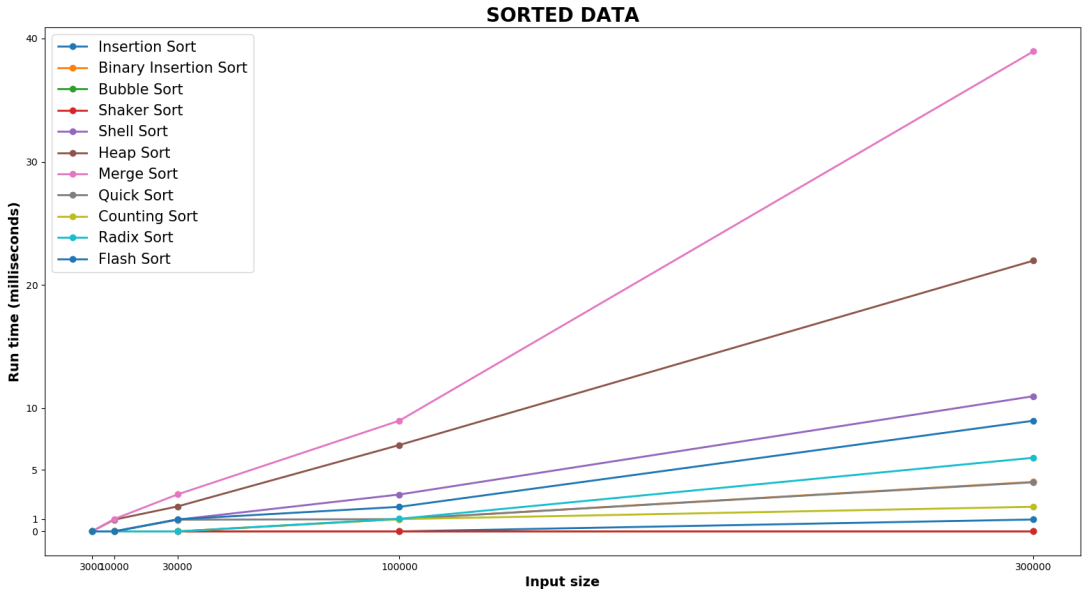
- Counting sort là thuật toán có thời gian thực thi nhanh nhất trong tất cả, với thời gian thực nghiệm cho bộ dữ liệu kích thước 300000 chỉ $3ms$, bởi với bộ dữ liệu có giá trị các phần tử nhỏ chỉ thuộc đoạn $[0, n-1]$, n tối đa là 300000, với bộ dữ liệu này, giải thuật có độ phức tạp thời gian $O(n)$ tuyến tính và nhanh hơn hẳn các thuật toán khác.

- Với thời gian thực thi rất nhanh, ngay sau Counting sort, là giải thuật Flash sort với thời gian thực nghiệm là $14ms$, và sau đó là Quick sort với $21ms$, Radix sort $25ms$. Đó là những giải thuật thuộc nhóm có độ phức tạp $O(nlog(n))$

2.2 Sorted data



Hình 2.2: Biểu đồ thời gian thực thi các thuật toán với bộ dữ liệu đã được sắp xếp



Hình 2.3: Biểu đồ thời gian thực thi các thuật toán tốt với bộ dữ liệu đã được sắp xếp

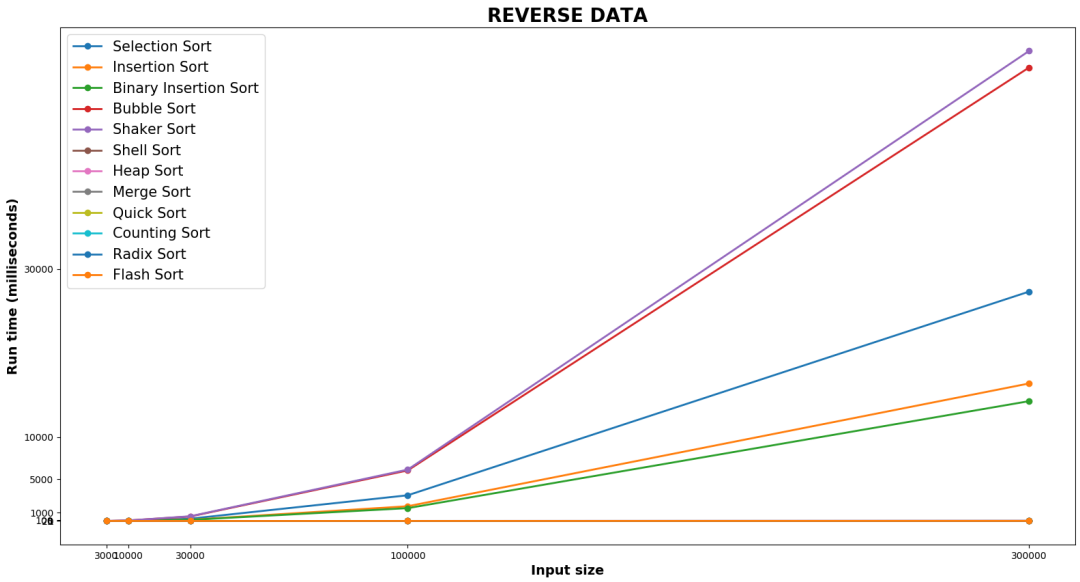
Nhận xét

- Trong trường hợp dãy có thứ tự sẵn, thuật toán Selection sort có thời gian thực thi chậm nhất ($26657ms$), trong khi các thuật toán có cùng độ phức tạp $O(n^2)$ khác lại có thời gian thực thi rất nhanh.

- Xét các thuật toán khác thuộc nhóm $O(n^2)$. Giải thuật Bubble sort và Shaker sort trong cài đặt để thực hiện thực nghiệm trên đã được cải tiến để dừng bước lặp sớm khi nhận thấy mảng đã có thứ tự. Insertion sort như đã phân tích ở mục 1.2.3. Với dữ liệu có thứ tự này là trường hợp mà các thuật toán trên hoạt động tốt nhất với độ phức tạp chỉ $O(n)$, thời gian thực thi chỉ khoảng $1ms$ và nhanh nhất với bộ dữ liệu có thứ tự này.

- Trong nhóm giải thuật $O(nlog(n))$, Quick sort là thuật toán có thời gian thực thi nhanh nhất ($4ms$), trong trường hợp dãy đã có thứ tự, ở mỗi lượt phân hoạch, Quick sort chọn pivot ở giữa đúng bằng trung vị của đoạn và rơi vào trường hợp tốt nhất. Và chậm nhất là giải thuật Merge sort ($39ms$)

2.3 Reverse data

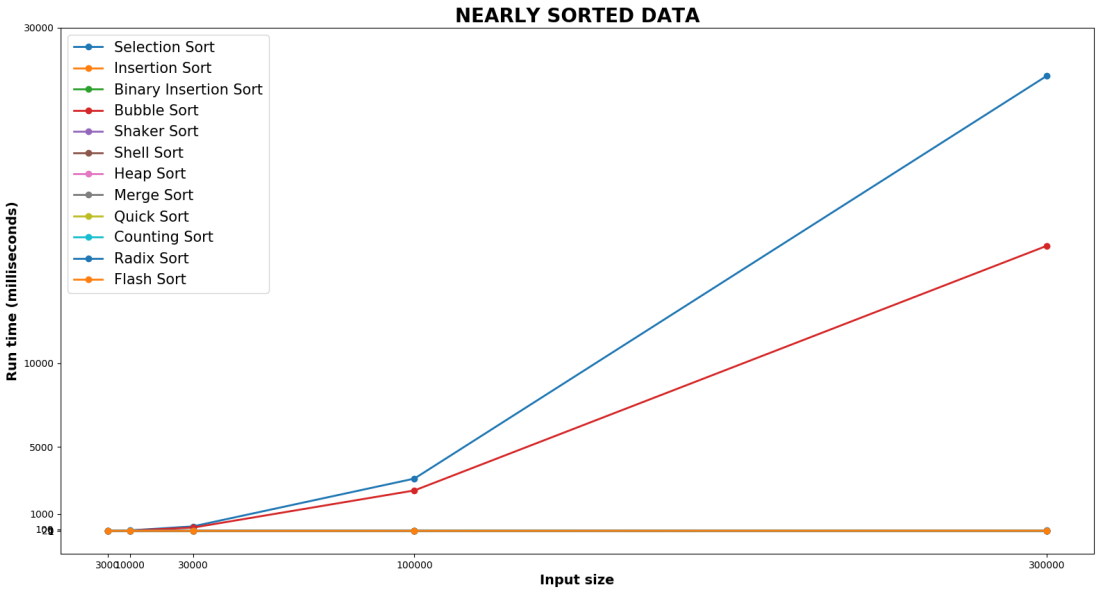


Hình 2.4: Biểu đồ thời gian thực thi các thuật toán với bộ dữ liệu thứ tự ngược

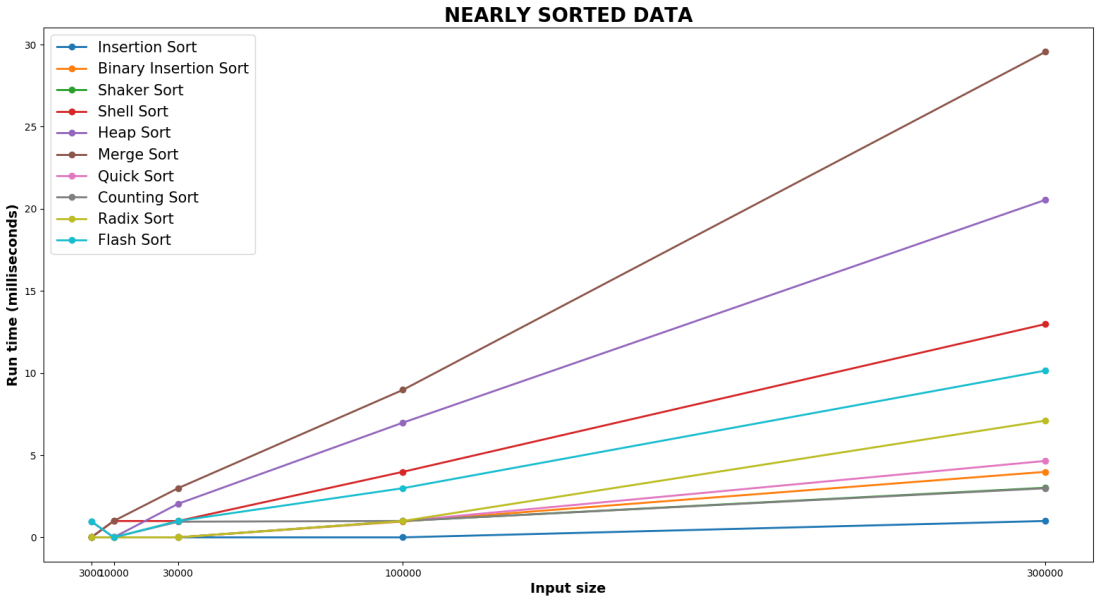
Nhận xét

- Trong trường hợp dãy có thứ tự ngược, Shaker sort và Bubble sort là hai thuật toán có thời gian thực thi chậm nhất ($50000ms$).
- Counting sort vẫn là thuật toán có thời gian thực thi nhanh nhất ($2ms$) ổn định với độ phức tạp thời gian $O(n)$
- Trong nhóm giải thuật $O(nlog(n))$, Quick sort là thuật toán có thời gian thực thi nhanh nhất ($3ms$), lý do ở mỗi bước phân hoạch, thuật toán chọn đúng trung vị của đoạn làm pivot, giúp Quick sort rơi vào trường hợp tốt nhất. Ngay sau đó là Radix sort ($5ms$). Và chậm nhất vẫn là Merge sort ($30ms$).

2.4 Nearly sorted data



Hình 2.5: Biểu đồ thời gian thực thi các thuật toán với bộ dữ liệu gần có thứ tự



Hình 2.6: Biểu đồ thời gian thực thi các thuật toán tốt với bộ dữ liệu gần có thứ tự

Nhận xét

- Đối với dãy gần có thứ tự, Selection sort là thuật toán có thời gian thực thi chậm nhất ($30000ms$), sau đó là Bubble sort ($17000ms$).
- Trong khi giải thuật Insertion sort tỏ ra rất hiệu quả với thời gian thực thi chỉ khoảng $1ms$ và là nhanh nhất trong bộ dữ liệu này. Lý do, khi dãy gần có thứ tự, thao tác so sánh được thực hiện rất ít và sớm kết thúc xấp xỉ với trường hợp tốt nhất $O(n)$ khi trên dãy đã có thứ tự.

- Trong nhóm giải thuật độ phức tạp $O(nlog(n))$, Quick sort vẫn giữ tốc độ thực thi tốt nhất với thời gian $5ms$. Chậm nhất vẫn là Merge sort ($30ms$)

3. Kết luận chung

3.1 Tính hiệu quả

Nhìn chung, các thuật toán có độ phức tạp thời gian $O(n^2)$ so với những giải thuật có độ phức tạp thời gian $O(nlog(n))$ đối với kích thước dữ liệu đầu vào nhỏ thì chênh lệch thời gian là không đáng kể, nhưng khi kích thước dữ liệu càng lớn thì các giải thuật $O(n^2)$ có thời gian thực thi tăng rất nhanh theo hàm bậc hai so với các giải thuật $O(nlog(n))$ chỉ tăng rất chậm. Ví dụ như khi xét trên bộ dữ liệu sinh ngẫu nhiên kích thước 300000 phần tử, giải thuật Quick sort chỉ chạy trong $20ms$ nhanh gấp 5000 lần so với Bubble sort chạy đến 10^5ms.

Dựa trên độ phức tạp thời gian, có thể chia các giải thuật sắp xếp thành 3 nhóm như sau:
Nhóm thuật toán sắp xếp có độ phức tạp $O(n^2)$

- Là các thuật toán như: *Selection sort, Insertion sort, Bianry-Insertion sort, Shaker sort*.
- Selection sort là thuật toán có độ phức tạp tính toán không phụ thuộc vào dữ liệu đầu vào. Trong khi các thuật toán như Bubble sort, Shaker sort hay Insertion sort thì có thời gian thực thi nhanh hay chậm phụ thuộc vào dữ liệu đầu vào.
- Nhìn tổng quan, giải thuật Insertion sort là giải thuật có thời gian thực thi nhanh hơn so với các thuật toán còn lại. Binary insertion sort tuy là phiên bản nâng cấp của Insertion sort nhưng trên thực tế không nhanh hơn nhiều lắm.
- Giải thuật Bubble sort là thuật toán hoạt động kém hiệu quả nhất vì tốn rất nhiều bước so sánh và hoán đổi vị trí.

Nhóm thuật toán sắp xếp có độ phức tạp $O(nlog(n))$

- Là các thuật toán như: *Heap sort, Merge sort, Quick sort*
- Nhìn tổng quan, Quick sort là thuật toán sắp xếp có thời gian thực thi nhanh nhất đúng như tên gọi của nó, để Quick sort hoạt động hiệu quả, ta nên chọn pivot làm trung vị hoặc chọn ngẫu nhiên, không nên chọn pivot ở đầu hoặc cuối mỗi đoạn để tránh trường hợp xấu nhất.
- Merge sort là thuật toán có thời gian thực thi chậm nhất nhưng có thời gian chạy tương đối ổn định khi luôn giữ thời gian thực thi trong khoảng $30ms - 40ms$ với bộ dữ liệu 300000 phần tử, lý do như phân tích ở mục $1.8.3$, Merge sort hoạt động không phụ thuộc vào phân bố dữ liệu đầu vào, ngoài ra còn phải tốn thêm không gian phụ.

Nhóm thuật toán sắp xếp khác

- Đây là nhóm các thuật toán sắp xếp không phụ thuộc vào thao tác so sánh trực tiếp các phần tử như: *Counting sort, Radix sort, Flash sort*.
- Counting sort là giải thuật hoạt động hiệu quả nhất trong tất cả các thuật toán được nêu trong báo cáo này. Bởi độ phức tạp tuyến tính $O(n)$ của nó khi hoạt động trên bộ dữ liệu mà các phần tử có giá trị nhỏ. Với bộ dữ liệu kích thước 300000 có thứ tự nào, Counting sort cũng cho ra thời gian thực thi chỉ trong khoảng $2ms - 3ms$.
- Flash sort hoạt động cũng rất hiệu quả, khi trong bộ dữ liệu ngẫu nhiên luôn có thời gian thực thi nhanh hơn Quick sort gần như gấp đôi.

3.2 Tính ổn định

- Một phương pháp sắp xếp được coi là ổn định nếu nó bảo toàn thứ tự ban đầu của các phần tử có giá trị bằng nhau trong dãy ban đầu.
- Trong những thuật toán được nghiên cứu ở trên, Bubble sort, Shaker sort, Selection sort, Insertion sort, Binary-Insertion sort, Counting sort, LSD Radix sort đều là những thuật toán sắp xếp có tính ổn định.
- Những thuật toán như Quick sort, Heap sort, Flash sort, MSD Radix sort, Shell sort, hay nói chung là các thuật toán sắp xếp phải sử dụng thao tác hoán đổi hai phần tử ở vị trí bất kì thì là không ổn định.