Trường Đại Học Khoa Học Tự Nhiên
Khoa Công Nghệ Thông Tin
Môn: Cấu trúc dữ liệu và giải thuật
Giáo viên hướng dẫn: Bùi Huy Thông

BÁO CÁO ĐỒ ÁN 1

PHÙNG THỊ HÒA -1512186

[Ý tưởng, thuật toán, độ phức tạp và đánh giá giải thuật của các thuật toán sắp xếp: selection sort, bubble sort, insertion sort, merge sort, quicksort, heap sort và radix sort. Trình bày về kết quả thực nghiệm về các thuật toán, đưa ra các nhận xét về kết quả.]

MỤC LỤC

Mục lục

GIẢI THUẬT SẮP XẾP

TỔNG QUAN GIẢI THUẬT SẮP XẾP

ĐỊNH NGHĨA

Sắp xếp là sắp xếp dữ liệu theo một định dạng cụ thể. Trong khoa học máy tính, giải thuật sắp xếp xác định cách sắp xếp dữ liệu theo một thứ tự nào đó. Sắp xếp theo thứ tự ở đây là sắp xếp theo thứ tự dạng số hoặc chữ cái trong từ điển.

CÔNG DỤNG

Công dụng của việc sắp xếp dữ liệu nằm ở chỗ: việc tìm kiếm dữ liệu được tối ưu nếu dữ liệu được sắp xếp theo thứ tự nào đó. Ngoài ra, sắp xếp giúp dữ liệu được định dạng dễ đọc hơn, sắp xếp các kết quả in ra trên bảng biểu ...

PHÂN LOẠI

- Sắp xếp cố định và sắp xếp so sánh.
- Sắp xếp in-place và not-in-place.
- Sắp xếp Adaptive và Non-Adaptive.

CÁC PHƯƠNG PHÁP SẮP XẾP THÔNG DỤNG

- Selection sort
- Bubble sort
- Insertion sort
- Shell sort
- Heap sort
- Quick sort
- Merge sort
- Radix sort
- Interchange sort
- Binary Insertion sort
- Shaker sort

GIẢI THUẬT SẮP XẾP

SELECTION SORT

Ý TƯỞNG

Selection sort là một trong những thuật toán sắp xếp đơn giản nhất. Ý tưởng cơ bản của cách sắp xếp này là:

- Lượt đầu tiên, ta chọn phần tử nhỏ nhất trong mảng từ vị trí 0 đến cuối mảng và đổi giá trị của phần tử nhỏ nhất với phần tử ở vị trí 0.
- Ở lượt thứ hai, ta chọn phần từ nhỏ nhất trong mảng từ vị trí 1 đến cuối mảng, đổi giá trị phần tử nhỏ nhất với các phần tử ở vị trí 1.
- ...
- Ta tiến hành lặp lại thao tác cho đến hết mảng.

THUẬT TOÁN

1. Thiết lập i=0.
2. Tìm phần tử nhỏ nhất trong dãy hiện hành tử a[i] đến a[n-1].
3. Hoán vị phẩn tử nhỏ nhất với a[i];
4. Nếu i<n-1 thì i++ và lặp lại bước 2.
   Ngược lại, dừng thuật toán.

VÍ DỤ

Figure 1: Mảng chưa được sắp xếp[1].

Figure 2: Tìm được phần tử nhỏ nhất là 10[2].

Figure 3: Hoán vị 10 và 14[3].

Figure 4: Thực hiện tương tự đến hết mảng[4].

CÀI ĐẶT

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

ĐỘ PHỨC TẠP THUẬT TOÁN

Với hai vòng lặp lồng vào nhau, ta ước lượng độ phức tạp của thuật toán này là $O(n^2)$ trong mọi trường hợp tốt nhất, xấu nhất và trung bình.

ĐÁNH GIÁ ĐỘ PHỨC TẠP

Đối với selection sort, ở mỗi lượt thứ i thì cần (n-i) lần phép so sánh. Số lượng phép so sánh không phụ thuộc vào tình trạng dãy số, do vậy trong mọi trường hợp số phép so sánh là:

$$\text{Phép so sánh} = \sum_{i=0}^{n-2}(n - i) = \frac{n(n-1)}{2}$$

Dựa vào số phép so sánh, ta kết luận độ phức tạp của thuật toán trong mọi trường hợp là $O(n^2)$.

BUBBLE SORT

Ý TƯỞNG

Chúng ta bắt đầu duyệt từ đầu mảng lên cuối, nếu gặp hai phần tử kế cận bị ngược thứ tự thì ta đổi chỗ chúng cho nhau. Sau lần duyệt như vậy, phần tử lớn nhất trong mảng sẽ được di chuyển về vị trí cuối và vấn đề tiếp theo trở thành sắp xếp dãy từ a[0] đến a[n-2]. Lặp lại các bước như vậy cho đến khi không còn phần tử để xét.

THUẬT TOÁN

1. i=0.
2. j=0;
   Khi (j<(n-i-1)) thực hiện:
   Nếu (a[j]>a[j+1]) thì hoán vị a[j] và a[j+1] .
   Tăng j: j++;
3. i++;
   Nếu i>=n-1 dừng.
   Ngược lại: Lặp lại bước 2.

VÍ DỤ

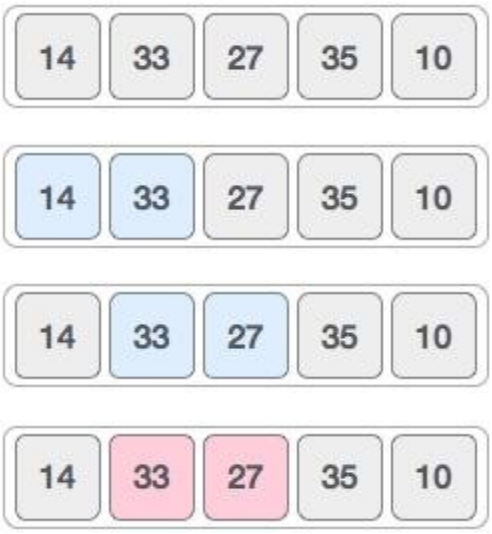


Figure 5: Ví dụ về cách thực hiện của Bubble Sort[5]

CÀI ĐẶT

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

ĐỘ PHỨC TẠP THUẬT TOÁN

Trong mọi trường hợp độ phức tạp của thuật toán luôn là $O(n^2)$ ngay cả khi mảng đã được sắp xếp.

Nhưng chúng ta có thể tối ưu được nó nếu chúng ta thực hiện dừng thuật toán nếu trong vòng lặp không có hoán vị. Lúc này độ phức tạp của thuật toán trong từng trường hợp được tối ưu sẽ là:

Trường hợp	Độ phức tạp
Tốt nhất	O(n)
Trung bình	$O(n^2)$
Xấu nhất	$O(n^2)$

ĐÁNH GIÁ ĐỘ PHỨC TẠP

- Số phép so sánh trong giải thuật này là không đổi, không phụ thuộc vào tình trạng ban đầu của dãy. Với i bấy kì, ta luôn phải so sánh a[j] và a[j+1], vì thế mà ta tốn n-i phép so sánh. Thêm vào đó là i chạy từ 0 đến (n-2). Vậy ta tính được tổng số phép so sánh ở đây là n(n-1)/2.
- Một phép hoán vị tương đương với ba phép gán và nó phụ thuoccj vào tình trạng ban đầu mảng được truyền vào. Cụ thể là:

  Ở trường hợp tốt nhất: Là trường hợp dãy ban đầu đã có thứ tự. Ta không cần tốn bất kỳ phép hoán vị nào.

  Ở trường hợp xấu nhất: Dãy có thứ tự ngược. Số phép hoán vị là

  n(n-1)/2.

NHẬN XÉT

Ở trong đồ họa máy tính, Bubble sort rất phổ biến vì khả năng phát biện ra một lỗi nhỏ (như đổi chỗ hai phần tử) ở một mảng gần như được sắp xếp và sửa nó với linear complexity là 2n.

INSERTION SORT

Ý TƯỞNG

Xét mảng a[0...n]. Ta thấy dãy con chỉ gồm một phần tử a[0] có thể coi là một dãy đã sắp xếp rồi. Xét thêm a[1], ta so sánh giữa a[0] và a[1], nếu thấy a[1]<a[0] thì ta chèn nó trước a[0]. Đối với a[2], ta chỉ dãy gồm a[0],a[1] đã được sắp xếp, ta tìm cách chèn a[2] sao cho phù hợp nhất để được thứ tự sắp xếp. Nói một cách tổng quát ta xếp mảng a[0...i] trong điều kiện mảng a[0...i-1] đã được sắp xếp rồi, bằng cách chèn a[i] vào chị trí đúng, để được toàn bộ mảng được sắp xếp phù hợp.

THUẬT TOÁN

1. i=1;
2. key=a[i]; pos=i-1;
3. Nếu (pos>=0 và a[pos]>key) tiến hành lặp: a[pos+1]=a[pos] và pos--;
4. a[pos+1]=key;
5. i++;

   Nếu i<n: Lặp lại bước 2.

   Ngược lại: Dừng.

CÀI ĐẶT

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

VÍ DỤ

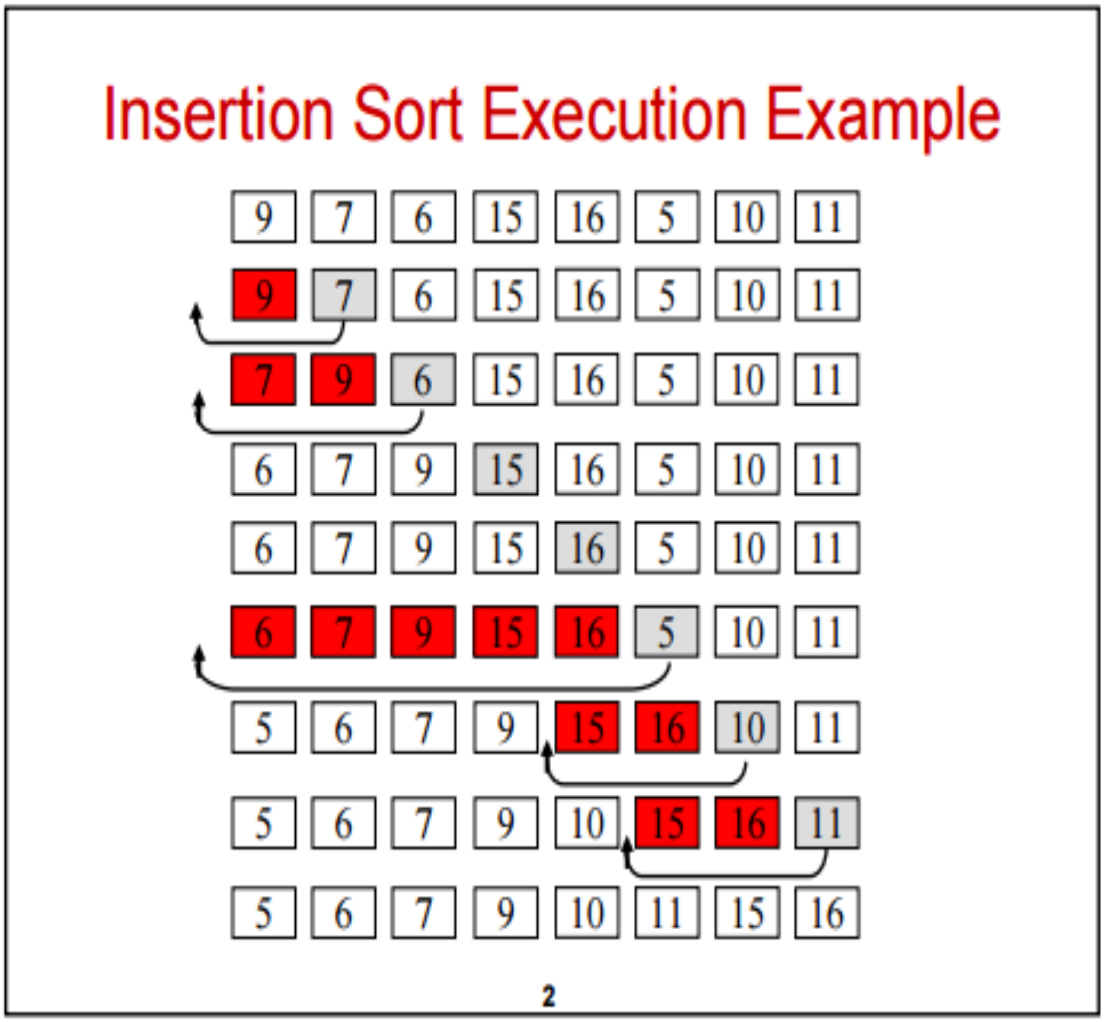


Figure 6: Cách thực thi của Insertion sort[6]

ĐỘ PHỨC TẠP THUẬT TOÁN

Độ phức tạp của thuật toán là $O(n^2)$.

ĐÁNH GIÁ ĐỘ PHỨC TẠP THUẬT TOÁN

- Trường hợp tốt nhất: Dãy ban đầu đã có thứ tự. Ta tìm được ngay vị trí thích hợp để chèn ngay lần so sánh đầu tiên mà không cần phải vào vòng lặp. Như vậy, với i chạy từ 1 đến n-1 thì số phép so sánh tổng cộng là n-1. Còn số với số phép gán, do thuật toán không chạy vào vòng lặp while nên chỉ thực hiên hai phép gán là key=a[i] và a[pos+1]=key. Vậy số phép gán bằng 2(n-1).
- Trường hợp xấu nhất: Dãy ban đầu có thứ tự ngược. Ta thấy ngay vị trí thích hợp pos luôn là vị trí đầu tiên của dãy đã có thứ tự, do đó, để tìm được vị trí này ta phải duyệt hết dãy đã có thứ tự. Với i bất kì, số phép so sánh là i-1 và số phép gán là i+1. Với i chạy từ 1 dến n-1, ta tính được số phép so sánh tổng cộng là n(n-1)/2 và số phép gán là [n(n-1)/2]-1.

Trường hợp	Độ phức tạp
Tốt nhất	$O(n)$
Xấu nhất	$O(n^2)$

NHẬN XÉT

Insertion sort được sử dụng khi số phần tử cần sắp xếp nhỏ. Nếu sắp xếp mảng lớn thì Insertion Sort không hiệu quả. Nó rất có ích khi dữ liệu đầu vào là mảng hầu như đã được sắp xếp hết, chỉ có một vài phần tử không đúng chỗ trong một mảng lớn. Nhìn vào kết quả, ta có thể nhận thấy rằng thuật toán Insertion sort tỏ ra tốt hơn rất nhiều so với thuật toán Selection sort và Bubble sort. Tuy nhiên chi phí thời gian còn rất lớn. Trên phương diện lý thuyết insertion sort vẫn có độ phức tạp là $O(n^2)$.

Ở đây ta có thể thực hiện cải tiến thuật toán bằng cách: Khi dãy a[0...i-1] đã được sắp xếp thì việc chèn có thể làm bằng thuật toán tìm kiếm nhị phân và kỹ thuật chèn có thể làm bằng các lệnh dịch chuyển vùng nhớ cho nhanh. Tuy nhiên điều đó nó không độ phức tạp thuật toán trở nên tốt hơn vì trong trong trường hợp xấu nhất, ta mất n-1 lần chèn và lần chèn thứ i ta phải dịch lùi i khóa để tạo ra khoảng trống trước khi đẩy vị trị cần chèn vào chỗ trống đó.

MERGE SORT

Ý TƯỞNG

Merge sort sử dụng giải thuật chia để trị (Divide and Conquer). Chúng ta tiến hành chia dữ liệu đầu vào làm hai phần, tiến trình này diễn ra cho tới khi không còn chia được nữa. Lúc này coi mỗi phần tử là một mảng riêng lẻ đã được sắp xếp. Hàm merge() sẽ được viết để tiến hành trộn hai mảng đã sắp xếp. Hàm merge(a,l,m,r) là chìa khóa để a[l..m] và a[m+1..r] được sắp xếp và trộn hai mảng này thành một.

THUẬT TOÁN

MergeSort(a,l,r)

Nếu (r>l) thì:

1. Tìm điểm nằm giữa (m) để chia mảng thành hai phần: m=(l+r)/2.
2. Gọi hàm MergeSort cho phần thứ nhất : MergeSort(a,l,m).
3. Gọi hàm MergeSort cho phần thử hai: MergeSort(a,m+1,r).
4. Gọi hàm merge cho hai phần đã được sắp xếp ở bước 2 và 3: merge(a,l,m,r).

CÀI ĐẶT

```
void merge(int * a, int l, int m, int r) {
	int i, j, k;
	int n1 = m - l + 1;
	int n2 = r - m;
	int*L = new int[n1];
	int*R = new int[n2];
	for (i = 0; i < n1; i++)
		L[i] = a[l + i];
	for (j = 0; j < n2; j++)
		R[j] = a[m + j + 1];
	i = 0;
	j = 0;
	k = 0;
	while (i < n1 && j < n2) {
		if (L[i] <= R[j]) {
			a[k] = L[i];
```

```
                    i++;
            }
            else {
                    a[k] = R[j];
                    j++;
            }
            k++;
      }
      while (i < n1) {
            a[k] = L[i];
            i++;
            k++;
      }
      while (j < n2) {
            a[k] = R[j];
            j++;
            k++;
      }
}
```

```
void mergeSort(int * a, int l, int r){
      if (l < r) {
            int m = l + (r - l) / 2;
            mergeSort(a, l, m);
            mergeSort(a, m + 1, r);
            merge(a, l, m, r);
      }
}
```

VÍ DỤ

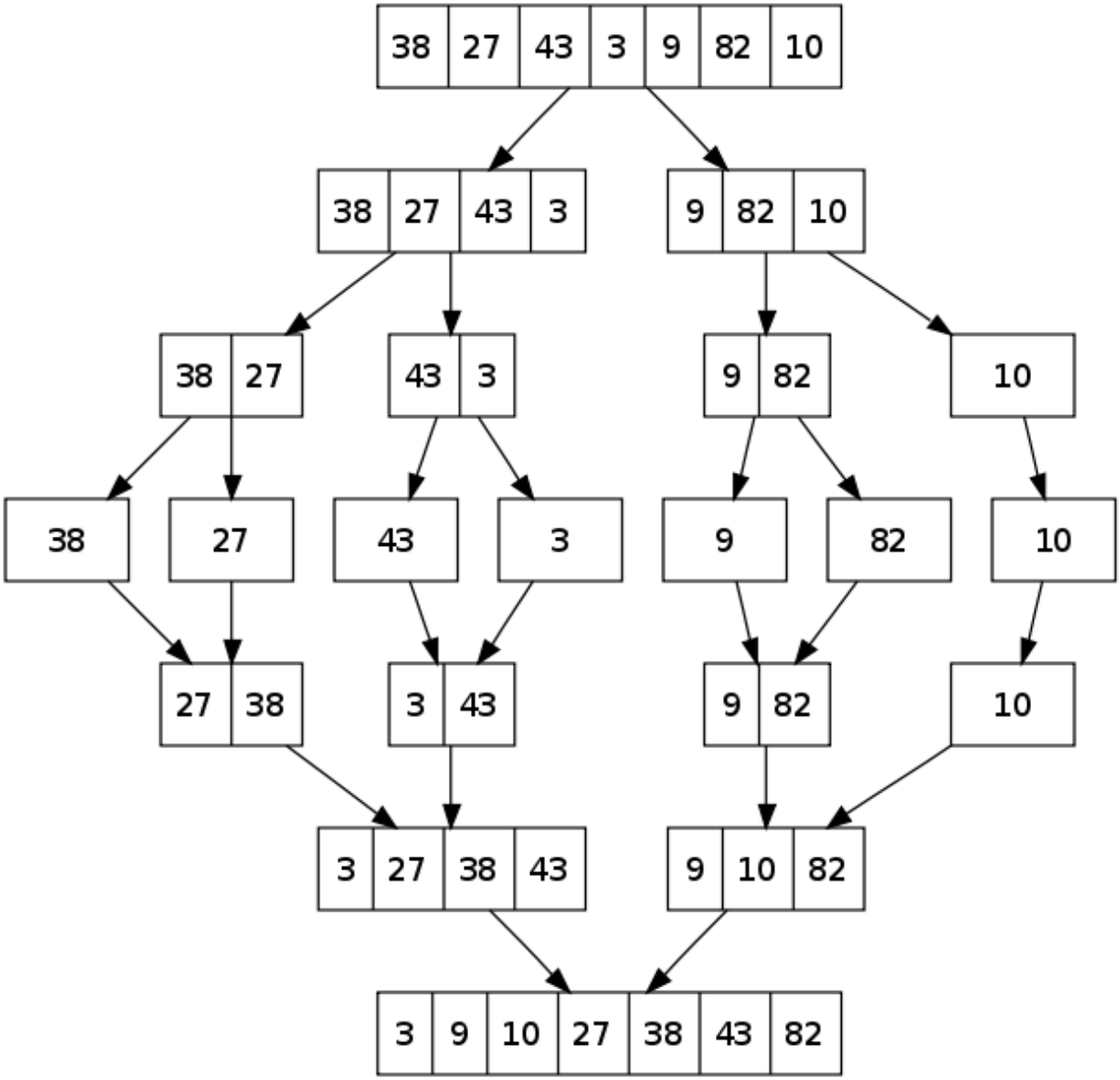


Figure 7: Ví dụ về cách thực thi của Merge Sort[7].

ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Độ phức tạp không đổi trong cả ba trường hợp xấu nhất, trung bình và tốt nhất đều là $O(n\log_2 n)$.

ĐÁNH GIÁ ĐỘ PHỨC TẠP

Merge Sort không sử dụng thông tin về đặc tính của dãy cần được sắp sếp nên trong mọi trường hợp của thuật toán chi phí là không đổi.

NHẬN XÉT

Một trong những nhược điểm của Merge sort là không tận dụng những thông tin về đặt tính của mảng cần được sắp xếp. Trong trường hợp dãy đã có thứ tự, hay không có thứ tự thời gian thực hiện sắp xếp là như nhau. Chính vì vậy trong thực tế phương pháp trộn trực tiếp ít được sử dụng, mà người ta tiến hành cái tiến nó bằng thuật toán trộn tự nhiên Natural Merge Sort.

So với Selection sort, Bubble sort, Insertion sort thì Merge sortn vẫn tối ưu hơn. Và nhanh hơn Quick sort trong một số trường hợp.

Ứng dụng của Merge sort:

- Merge sort rất hữu ích trong việc sắp xếp danh sách liên kết với độ phức tạp về thời gian là $O(n\log_2 n)$.
- Sử dụng cho cả External Sorting

QUICK SORT

Ý TƯỞNG

Sắp xếp mảng a[0..n-1] thì có thể coi là sắp xếp đoạn từ chỉ số 0 đến n-1 trong mảng đó. Để sắp xếp một đoạn trong dãy khóa, nếu đoạn đó có <= 1 phần tử thì không cần phải làm gì cả, còn nếu đoạn đó có ít nhất 2 phần tử, ta chọn phần tử ở vị trí giữa mảng đó để làm chốt(pivot). Mọi phần tử nhỏ hơn chốt được sắp xếp vào vị trí trước chốt, mọi phần tử lớn hơn chốt sẽ được xếp vị trí sau chốt. Sau phép hoán vị như vậy thì đoạn đang xét được chia làm hai đoạn rỗng mà mọi phần tử trong đoạn đầu đều bé hơn hoặc bằng chốt, mọi phần tử ở đoạn sau đều lớn hơn hoặc bằng chốt. Vấn đề trở thành sắp xếp hai đoạn mới tạo ra bằng cách lặp lại việc tương tự. Ta cứ tiến hành cho tới khi mảng chia thành các mảng có có độ dài bằng 1.

THUẬT TOÁN

void quickSort(int*a, int low, int high):

1. Lấy phần tử ở vị trí (low+high)/2 làm mốc(pivot).
2. Chia mảng bằng với phần tử chốt.
3. Gọi quickSort với mảng con bên trái.
4. Gọi quickSort với mảng con bên phải.

CÀI ĐẶT

```
void quickSort(int * a, int low, int high){
	if (low >= high) return;
	int pivot = a[(low + high) / 2];
	int i = low, j = high;
	do {
		while (a[i] < pivot) ++i;
		while (a[j] > pivot) --j;
		if (i <= j) {
			if (i < j) swap(a[i], a[j]);
			++i;
			--j;
```

```
            }
        } while (i <= j);
        quickSort(a, low, j);
        quickSort(a, i, high);
}
```

VÍ DỤ

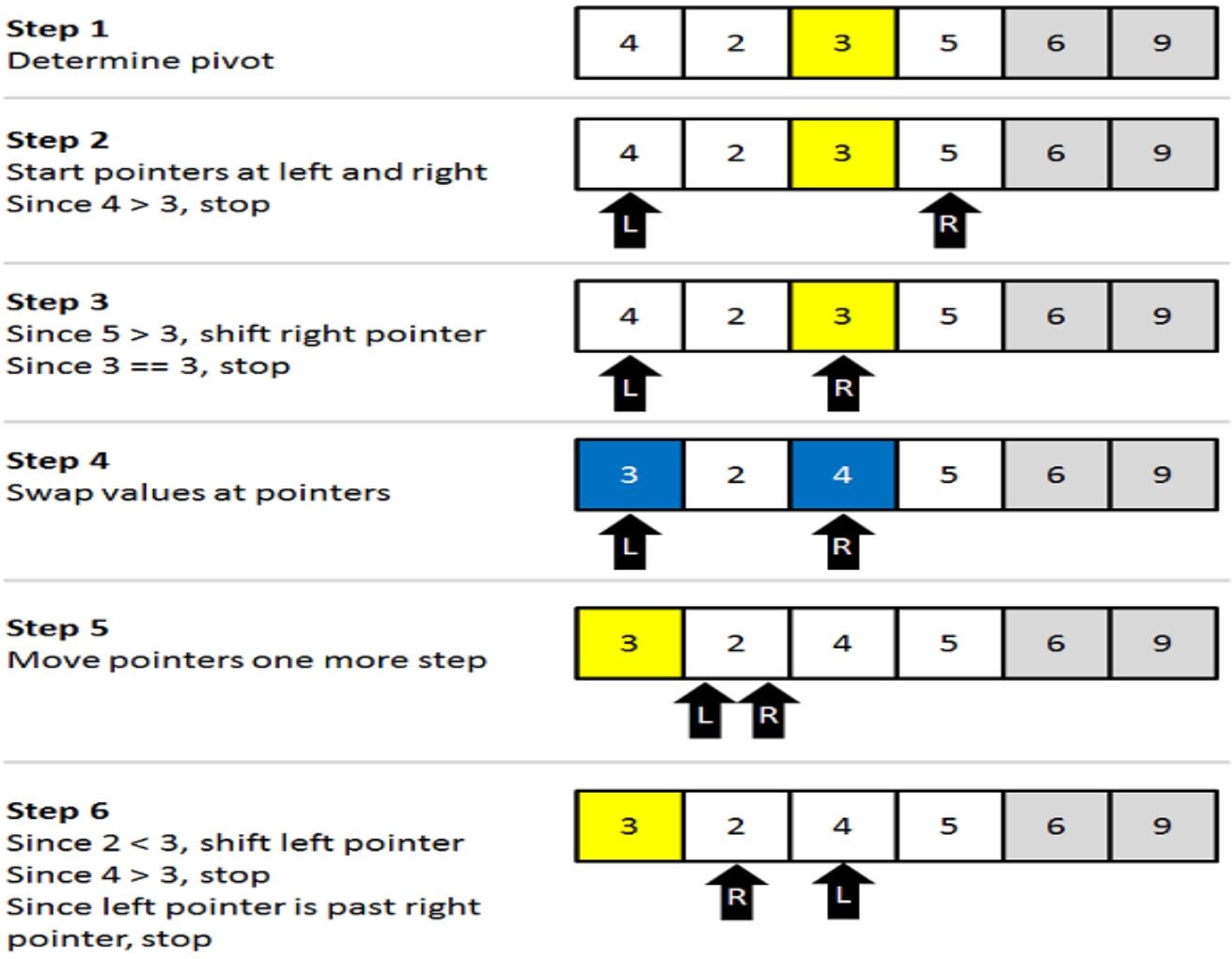


Figure 8: Hình minh họa cách thức hoạt động của Quick Sort[8]

ĐỘ PHỨC TẠP THUẬT TOÁN

Trường hợp	Độ phức tạp
Tốt nhất	$O(n\log_2 n)$
Trung bình	$O(n\log_2 n)$
Xấu nhất	$O(n^2)$

ĐÁNH GIÁ GIẢI THUẬT

Ta thấy được hiệu quả của thuật toán phụ thuộc vào việc chon giá trị làm phần tử chốt.

- Trường hợp tốt nhất: Mỗi lần phân hoạch để chọn được phần tử từ median(phần tử lớn hơn hay bằng nửa số phần tử và nhỏ hơn hay bằng nửa số phần tử còn lại) làm mốc. Khi đó dãy được phân hoạch thành hai phần bằng nhau, ta cần $\log_2(n)$ lần phân hoạch thí sắp xếp xong. Ta cũng dễ thấy trong mỗi lần phân hoạc ta cần duyệt qua n phần tử. Vậy độ phức tạp trong trường hợp tốt nhất là $O(n\log_2 n)$.
- Trường hợp xấu nhất: Mỗi lần phân hoạch ta chọn phải phần tử có giá trị cực đại hoặc cực tiểu làm mốc. Khi đó dãy bị phân hoạch thành hai phần không đều nhau: Một phần chỉ có một phần tử, phần kia có n-1 phần tử. Do đó ta cần tới n lần phân hoạch mới sắp xếp xong. Vậy độ phức tạp trong trường hợp này là $O(n^2)$.

NHẬN XÉT

Mặc dù trong trường hợp xấu nhất, Quick sort có độ phức tạp là $O(n^2)$, nó so với Merge sort và Heap sort thì lớn hơn nhiều. Nhưng trong thực tế thì Quick sort chạy nhanh và nó được sử dụng nhiều trong sắp xếp thực tế. Quick sort có thể thực thi với nhiều cách khác nhau bằng cách thay đổi các phần tử làm mốc.

HEAP SORT

Ý TƯỞNG

Heap sort dựa trên cấu trúc dữ liệu là Binary Heap. Nó giống như Selection Sort, nó tìm phần tử lớn nhất và đưa phần tử đó về cuối. Chúng ta lặp lại việc đó cho khi mảng đã được sắp xếp hết.

Vậy thế nào là Binary Heap? Đầu tiên ta hiểu rằng, Binary Heap là một cây nhị phân đầy đủ, nó có một thứ tự đặt biệt là giá trị của node cha lớn hơn (hoặc nhỏ hơn) giá trị của hai node con. Đó gọi là Max Heap nếu cha lớn hơn hai node con, Min Heap nếu node cha bé hơn hai node con. Heap có thể biểu diễ cây nhị phân hoặc mảng.

Từ một Binary Heap, chúng được biểu diễn như mảng. Nếu node cha đã được xếp tại vị trí i, node con trái được dự tính nằm ở vị trí 2*i+1 và node con phải nằm ở vị trí 2*i+2 (mảng bắt đầu từ vị trí 0).

THUẬT TOÁN

1. Xây dựng Max Heap từ dữ liệu đầu vào.
2. Khi xây dựng được Max Heap, phần tử lớn nhất là gốc của Heap. Đổi chỗ nó với phần tử cuối cùng của Heap. Lúc này kích thước của Heap giảm đi 1. Gọi hàm heapify để đẩy giá trị nút gốc xuống vị trí phù hợp và đảm bảo tính chất của Heap.
3. Lặp lại các bước trên cho tới khi kích thước của Heap lớn hơn 1.

CÀI ĐẶT

```
void heapify(int * a, int n, int i){
	int largest = i;
	int l = 2 * i + 1;
	int r = 2 * i + 2;
	if (l<n && a[l]>a[largest])
		largest = l;
	if (r<n && a[r]>a[largest])
		largest = r;
```

```
        if (largest != i) {
                swap(a[i], a[largest]);
                heapify(a, n, largest);
        }
}
```

```
void heapSort(int * a, int n){
        // build heap
        for (int i = n / 2; i >= 0; i--)
                heapify(a, n, i);
        for (int i = n - 1; i >= 0; i--) {
                swap(a[0], a[i]);
                heapify(a, i, 0);
        }
}
```

ĐÁNH GIÁ ĐỘ PHỨC TẠP

Độ phức tạp thời gian của thao tác heapify là $O(\log_2 n)$. Độ phức tạp thời gian của tạo heap là $O(n)$. Vậy nên độ phức tạp về mặt thời gian của Heap Sort là $O(n\log_2 n)$.

NHẬN XÉT

Ứng dụng của Heap sort:

- Sắp xếp một mảng đã có thứ tự gần đúng.
- Tìm k phần tử lớn nhất hoặc bé nhất trong mảng.

RADIX SORT

Ý TƯỞNG

Khác với các thuật toán trước cơ sở sắp xếp chủ yếu là dựa trên việc so sánh hau giá trị phần tử thì Radix sort là thuật toán tiếp cận theo hướng khác, dựa trên nguyên tắc loại thư của bưu điện. Nó không quan tâm đến việc so sánh phần tử mà nó thực hiện việc phân loại, và từ việc phân loại sẽ tạo ra thứ tự cho các phần tử.

THUẬT TOÁN

1. Viết hàm int getMax(int*a, int n) để tìm phần tử lớn nhất để biết số phân phối hiên hành. Vì dụ 1000 là phần tử lớn nhất trong mảng, thì sẽ có 4 đơn vị hiện hành đó là: đơn vị, chục, trăm, nghìn.
2. Viết hàm void countSort (int*a, int n, int exp): để thực hiện đếm phân phối để sắp xếp các số theo giá trị hàng đơn vị, hàng trăm, hàng chục....
3. Lặp lại các bước trên cho các hàng cao hơn, cho đến khi không còn giá trị nào lớn hơn hàng ngang bằng đang xem xét. Cuối cùng ta thu được chuỗi đã được sắp xếp.

CÀI ĐẶT

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

```
void countSort(int * a, int n, int exp){
	int* output = new int[n];
	int i, count[10] = { 0 };
	for (i = 0; i < n; i++)
		count[(a[i] / exp) % 10]++;
	for (i = 1; i < 10; i++)
		count[i] += count[i - 1];
```

```
	for (i = n - 1; i >= 0; i--) {
		output[count[(a[i] / exp) % 10] - 1] = a[i];
		count[(a[i] / exp) % 10]--;
	}
	for (i = 0; i < n; i++)
		a[i] = output[i];
}
void RadixSor(int * a, int n){
	int m = getMax(a, n);
	for (int exp = 1; m / exp > 0; exp *= 10)
		countSort(a, n, exp);
}
```

ĐÁNH GIÁ ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Độ phức tạp của thuật toán là O(n).

KẾT QUẢ THỰC NGHIỆM

Kết quả thu được khi thực hiện lần lượt bảy thuật toán với kích thước dữ liệu đầu vào là 1000, 3000, 10000, 30000, 100000. Với 4 kiểu dữ liệu là Random, Sorted, Reverse, Near Sorted.

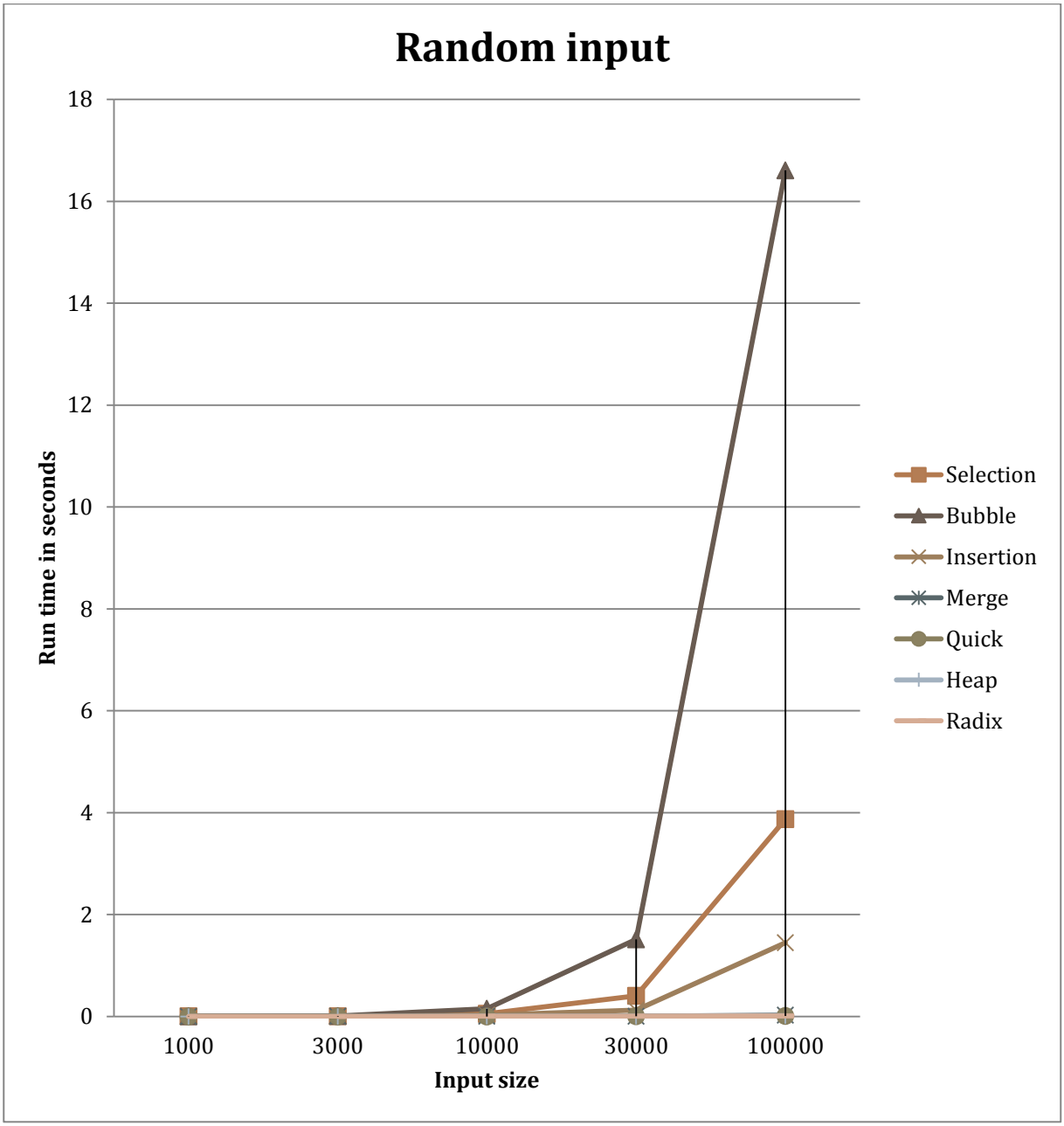
LƯU Ý: Kết quả thu được, khi chạy chương trình ở chế độ Release.

BẢNG KẾT QỦA THỰC NGHIÊM

Input State	Input Size	Selection	Bubble	Insertion	Merge	Quick	Heap	Radix
Random	1000	0.001	0.001	0.001	0	0	0	0
Random	3000	0.004	0.009	0.002	0.001	0	0	0
Random	10000	0.054	0.149	0.024	0.003	0	0.001	0.001
Random	30000	0.405	1.513	0.122	0.006	0.003	0.004	0.002
Random	100000	3.873	16.608	1.444	0.026	0.007	0.012	0.007
Sorted	1000	0	0	0	0.001	0	0	0.001
Sorted	3000	0.004	0.003	0	0	0	0.001	0
Sorted	10000	0.065	0.035	0	0.002	0	0	0.001
Sorted	30000	0.353	0.323	0	0.004	0	0.002	0.003
Sorted	100000	3.916	3.652	0	0.014	0.001	0.007	0.009
Reverse	1000	0	0.001	0.001	0.001	0	0	0
Reverse	3000	0.003	0.008	0.005	0.001	0	0	0.001
Reverse	10000	0.035	0.088	0.034	0.001	0	0.001	0.001
Reverse	30000	0.321	0.782	0.316	0.004	0.001	0.002	0.002
Reverse	100000	3.851	8.686	2.992	0.014	0	0.007	0.011
NearSorted	1000	0	0.001	0	0	0	0	0
NearSorted	3000	0.004	0.005	0	0.001	0	0	0
NearSorted	10000	0.052	0.035	0	0.001	0	0.001	0
NearSorted	30000	0.354	0.396	0	0.005	0	0.002	0.003
NearSorted	100000	3.737	3.929	0	0.02	0.002	0.008	0.013

DỮ LIÊU ĐẦU VÀO LÀ NGẪU NHIÊN

- Biểu đồ

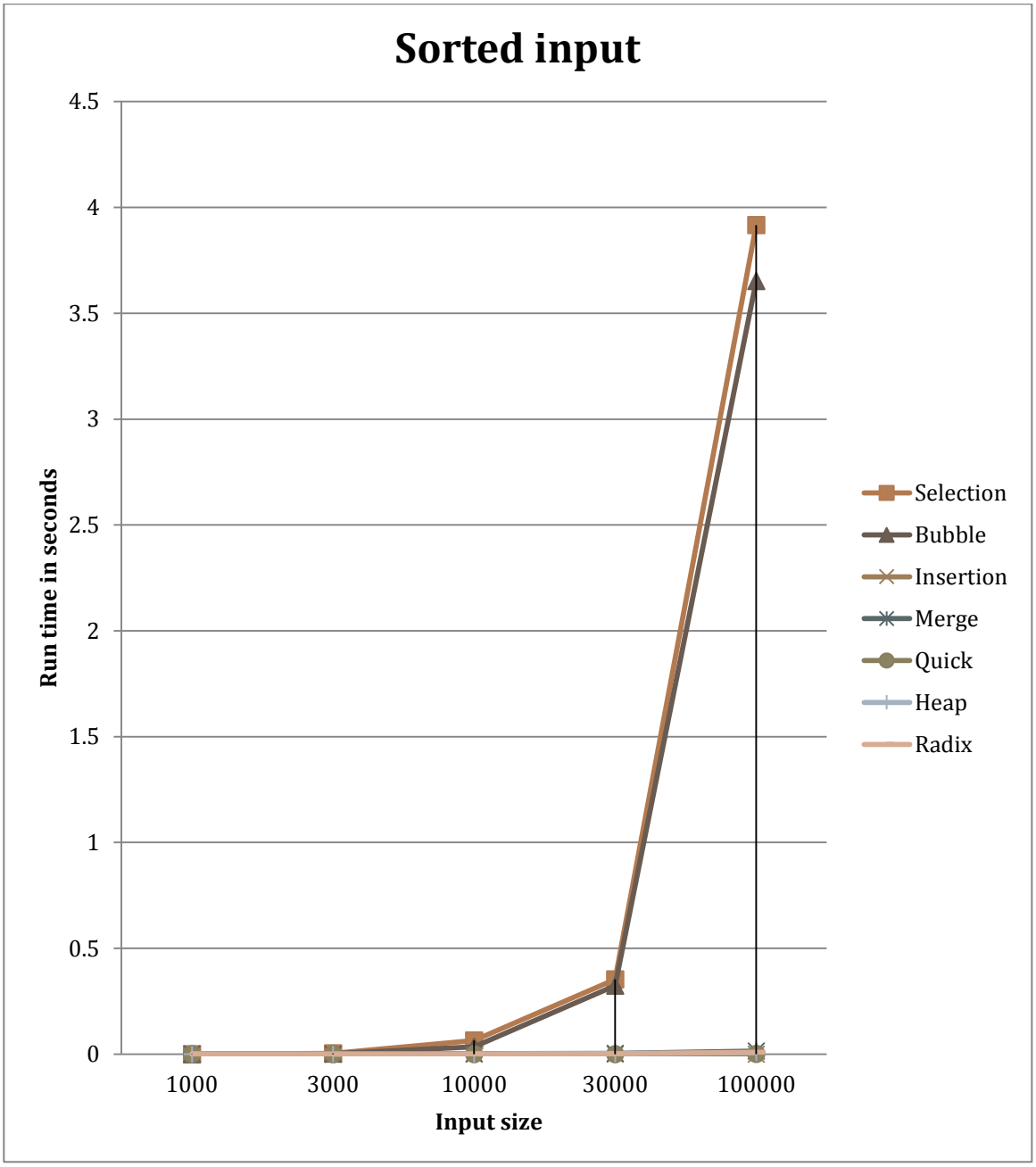


Biểu đồ 1: Biểu đồ khi dữ liệu đầu vào là Random.

- Nhận xét
  - ✓ Khi dữ liệu truyền vào có kích thước bé hơn 1000, các thời gian thực thi của các thuật toán gần như bằng 0. Lúc này ta chưa thấy hết được sự tối ưu của từng thuật toán. Nhưng khi bắt đầu nâng kích thước dữ liệu đầu lên đến 100000 phần tử thời gian thực thi bắt đầu thay đổi.
  - ✓ Đối với Merge sort, Quick sort, Heap sort, Radix sort thời gian thực thi tăng rất ít và không đáng kể. Thời gian thực thi của các thuật toán rất tối ưu. Merge sort có phần chậm hơn Quick, Heap và Radix sort.
  - ✓ Còn ba thuật toán còn lại là Bubble sort, Selection sort, Insertion sort thời gian thực thi tăng hàng chục hàng trăm lần. Dữ liệu càng lớn thời gian thực thi càng nhiều. Vì thế ta có thể kết luận, đối với loại dữ liệu có kích thước bé, việc lựa chọn thuật toán là Selection sort, Bubble sort, Insertion sort là có thể chấp nhận được, vì cách cài đặt của ba thuật toán này là khá đơn giản. Nhưng đối với dữ liệu có kích thước khổng lồ, sắp xếp sử dụng ba cách đó không được khả thi. Cần cân nhắc sử dụng các thuật toán khác tối ưu hơn đối với từng loại dữ liệu đầu vào.
  - ✓ Selection sort, Bubble sort, Insertion sort đều có độ phức tạp thuật toán là $O(n\log_2 n)$. Nhưng ta có thể thấy, khi thực thi, Bubble sort có thời gian thực thi nhiều nhất khi kích thước dữ liệu đầu vào là 100000. Nó gấp Selection sort gần 4 đến 5 lần, gấp Insertion sort 11.5 lần. Điều này xảy ra vì khi thực hiện sắp xếp, Bubble sort cần thực hiện hoán vị rất nhiều lần, điều này làm cho nó thực thi chậm hơn.

DỮ LIỆU ĐẦU VÀO ĐÃ ĐƯỢC SẮP XẾP

- Biểu đồ

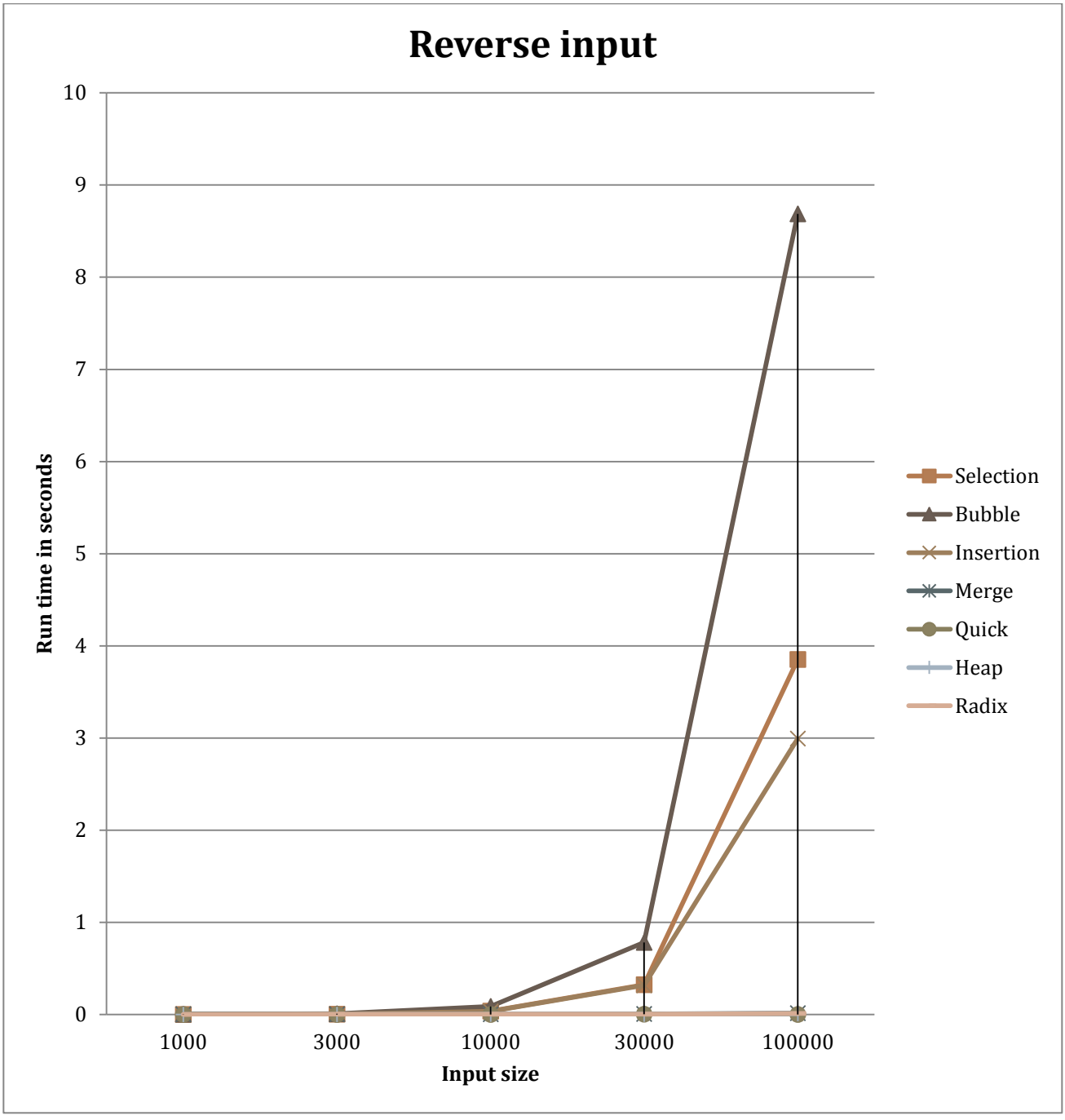


Biểu đồ 2: Biểu đồ khi dữ liệu đầu vào là Sorted.

- Nhận xét
  - ✓ Đối với dữ liệu đầu vào là mảng đã được sắp xếp, lúc này ta thấy thời gian thực thi của Insertion sort qua các lần tăng kích thước dữ liệu đầu vào đều bằng 0. Vì lúc này nó không tốn bất cứ chi phí nào cho việc dời vị trí. Vậy trường hợp tốt nhất của Insertion sort là khi mảng đầu vào đã được sắp.
  - ✓ Khi dữ liệu đã sắp xếp, thời gian thực thi của Bubble sort so với lúc dữ liệu truyền vào là ngẫu nhiên đã giảm xuống khoảng 4 đến 5 lần (dữ liệu truyền vào có kích thước 100000). Tuy nhiên, Bubble sort và Selection sort có thời gian thực thi so với các thuật toán khác vẫn lớn hơn rất nhiều.
  - ✓ Trong trường hợp mảng đã được sắp xếp hay ngẫu nhiên thì thời gian thực thi của Merge sort không có sự biến đổi nhiều. Vì thế đây là một điểm yếu của Merge sort, nó không phân biệt được thông tin và thuộc tính của mảng đầu vào.
  - ✓ Heap sort, Quick sort và Radix sort thực hiện rất tốt kể cả khi nâng kích thước dữ liệu. Khi cày đặt Quick sort, phần tử làm chốt là phần tử ở giữa vì thế trong trường hợp này, pivot rơi vào phần tử median nên thời gian thực thi nhanh hơn so với trường hợp dữ liệu đầu vào là ngẫu nhiên.

DỮ LIỆU ĐẦU VÀO LÀ CÓ THỨ TỰ NGƯỢC

- Biểu đồ

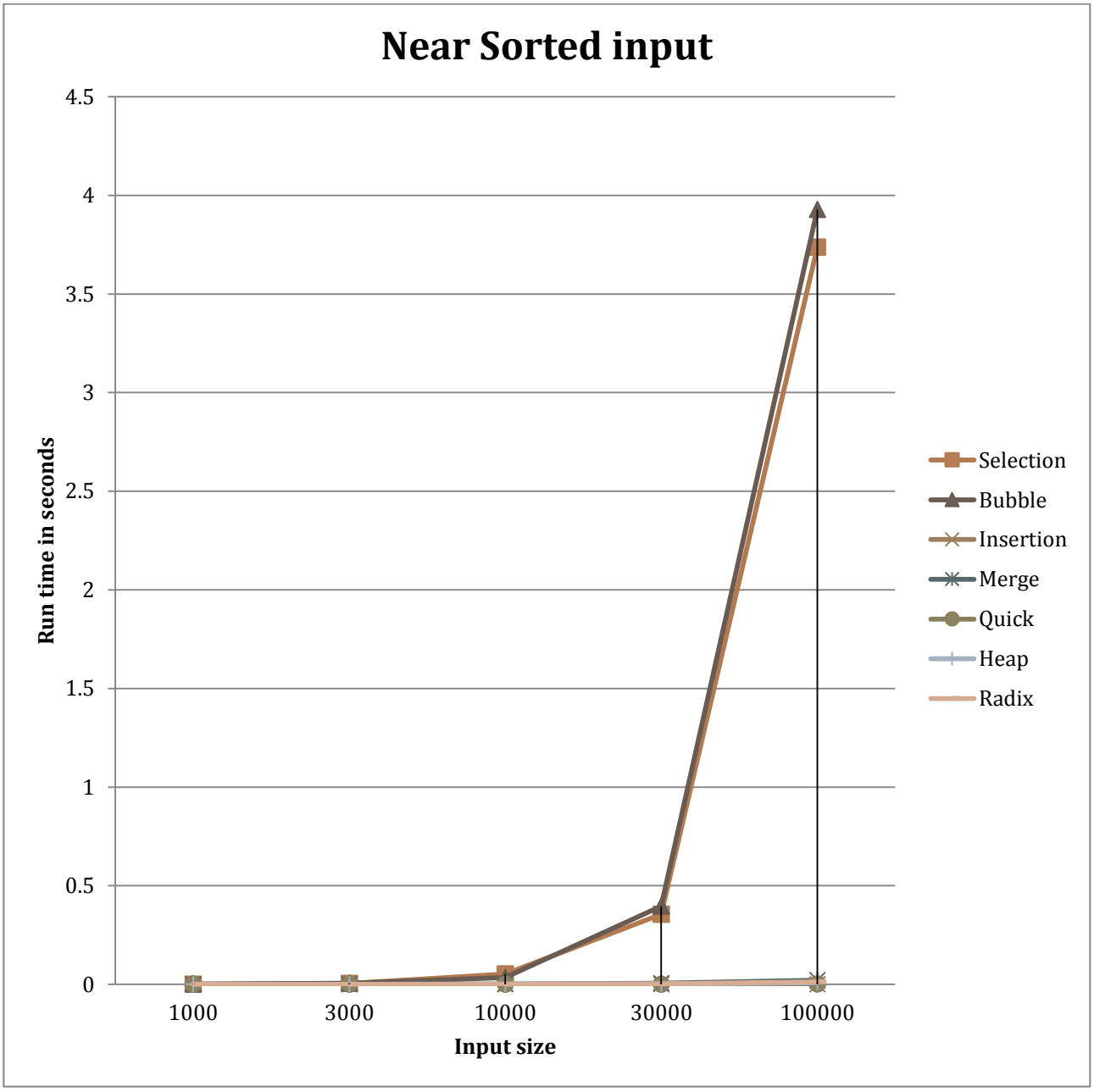


Biểu đồ 3: Biểu đồ khi dữ liệu đầu vào Reverse

- Nhận xét
  - ✓ Đối với Selection sort, so với các trường hợp dữ liệu đầu vào khác thì thời gian thực thi trung bình không chênh lệch bao nhiêu.
  - ✓ Trong trường hợp mảng bị đảo ngược thời gian thực thi của Bubble sort (8.686s) thời gian thực thi gần như gấp đôi so với mảng đầu vào đã được sắp xếp (3.652) khi size input=100000.
  - ✓ Cũng với độ phức tạp là $O(n^2)$, nhưng Insertion có thời gian thực thi bé hơn so với Bubble sort. Trong trường hợp mảng bị đảo ngược, thời gian thực thi của Insertion sort (2.992s)đây trường hợp Insertion sort có thời gian thực thi là lớn nhất so với các trường hợp dữ liệu đầu vào khác. Vậy khi mảng đảo ngược, thời gian thực thi của thuật toán này rơi vào tình trạng xấu nhất.
  - ✓ Merge sort có thời gian thực thi không thay đổi nhiều so với hai dữ liệu đầu vào trước.
  - ✓ Quick sort thời gian thực thi gần như bằng 0.
  - ✓ Heap sort và Radix có thời gian thực thi không thay đổi nhiều. Tốc độ vẫn rất nhanh và tối ưu.

DỮ LIỆU ĐẦU VÀO GẦN CÓ THỨ TỰ

- Biểu đồ



Biểu đồ 4: Biểu đồ khi dữ liệu đầu vào là Near Sorted

- Nhận xét
  - ✓ Thời gian thực thi của Selection sort và Bubble sort trong trường hợp này có kết quả gần giống như trường hợp dữ liệu đầu vào đã sắp xếp. Chênh lệch không lớn.
  - ✓ Insetion sort có kết quả bằng 0.
  - ✓ Merge sort hầu như không thay đổi nhiều so với các trường hợp trước.
  - ✓ Quick sort, Heap sort và Radix sort thực hiện rất nhanh. Nhưng xét tổng thời gian thực thi trung bình của 3 thuật toán này, Quick sort có kết quả bé nhất. Vậy so với hai thuật toán là Heap sort và Radix thì nhìn chung Quick sort vẫn tốt hơn.

Nhận xét chung:

- Selection sort là thuật toán dễ cài đặt, nhưng thời gian thực thi khá lớn, trong hầu hết các trường hợp dữ liệu đầu vào khác nhau thời gian thực thi không thay đổi nhiều.
- Bubble sort cũng là độ phức tạp $O(n^2)$ như Selection sort và Insertion sort nhưng thời gian thực thi của của nó là lớn nhất. Đặt biệt là trong trường hợp dữ liệu đầu vào ngẫu nhiên và bị đảo ngược. Khi mảng đã được sắp xếp thì thời gian thực thi bé nhất vì không tốn chi phí cho việc hoán đổi vị trí.
- Insertion sort độ phức tạp giống Selection và Bubble nhưng khi thời gian thực thi của nó khi dữ liệu đầu vào đã sắp xếp và gần sắp xếp đều bằng 0. Thời gian thực thi lớn nhất rơi vào trường hợp dữ liệu đầu vào bị đảo ngược.
- Merge sort trong mọi trường hợp dữ liệu đầu vào đều có thời gian thực thi không đổi vì nó không nhận diện được dữ liệu đầu vào, bất kì dữ liệu nào truyền vào cũng phải lần lượt thực hiện các thao tác trộn và sắp xếp một cách tuần tự.
- Heap sort, Radix sort và Quick sort có thời gian thực thi rất tốt. Nhưng vì thời gian thực thi của Quick sort phụ thuộc vào việc chọn pivot vì thế trong một số trường hợp phần tử pivot là median, nên thời gian thực thi rơi vào tình trạng tốt nhất, rất nhanh và tối ưu. Cả ba thuật toán này đều thực thi rất nhanh, nhưng xét về thời gian thực hiện trung bình, Quick sort vẫn có thời gian thực thi tốt nhất. Vì thế mà trong thực tế, Quick sort được sử dụng rất nhiều.

NGUỒN THAM KHẢO

- http://vietjack.com/
- https://www.hackerrank.com/
- https://www.tutorialspoint.com/
- http://www.geeksforgeeks.org/
- https://vi.wikipedia.org
- Một số hình ảnh, nguồn từ Internet.

GIẢI THUẬT SẮP XẾP