Độ phức tạp và giải thuật của các thuật toán sắp xếp

Trình bày : Trần Trọng Quyền

Tổng quan

1.Độ phức tạp tính toán

2.Các giải thuật sắp xếp

- 2.1 Selection Sort
- 2.2 Insertion Sort
- 2.3 Bubble Sort
- 2.4 Quick sort
- 2.5 Heap sort
- 2.6 Couting sort

3.Tìm kiếm trên cấu trúc mảng

- 3.1 Tìm kiếm nhị phân

1. Độ phức tạp của thuật toán

- Thuật toán là một phương pháp hướng dẫn để giải quyết một vấn đề hoặc thực hiện một nhiệm vụ cụ thể.
- Nó là một tập hợp các chỉ thị hoặc quy trình tính toán được sử dụng để giải quyết một vấn đề hoặc thực hiện một nhiệm vụ.

Đặc điểm của một thuật toán tốt

- **Đúng:** Thuật toán phải sản sinh ra kết quả chính xác cho mọi trường hợp.
- **Hiệu quả:** Thuật toán phải chạy nhanh và tiêu tốn ít tài nguyên.
- **Dễ hiểu:** Thuật toán phải được viết một cách dễ hiểu và dễ sửa đổi.
- **Tối ưu:** Thuật toán nên tối ưu hóa để đạt được kết quả tốt nhất với các ràng buộc và hạn chế.

Độ phức tạp của thuật toán

- **Độ phức tạp thời gian:** Đo lường số lần thực hiện các thao tác cơ bản mà thuật toán thực thi với tham biến n là kích thước đầu vào. y=f(n). Thực tế thường đo lường trường hợp xấu nhất của thuật toán vì sự đơn giản và thực tế của nó
- **Độ phức tạp không gian:** Đo lường lượng bộ nhớ cần thiết để thực hiện thuật toán.
- Độ phức tạp thường được biểu diễn bằng ký hiệu "O" (Big O), đại diện cho giới hạn trên của độ phức tạp.

Độ phức tạp BigO

Xét 2 hàm số dương $f(n)$ và $g(n)$ Ta ký hiệu: $f(n) = O(g(n))$

> Theo định nghĩa giải tích, ký hiệu trên tương đương với: $\lim_{n \to \infty} \sup \dfrac{f(n)}{g(n)} < \infty$ Gọi là "hàm f không tăng (tiệm cận) nhanh hơn g".

Nói một cách dễ hiểu: $f(n) = O(g(n))$ thì tồn tại hằng số $c > 0$ để khi n đủ to $($với mọi $n \geq n_0$ nào đó$)$ thì $f(n) \leq c \times g(n)$.

Ví dụ:

- $f(n) = 2n + 10$ là $O(n)$ vì khi chọn $c = 3$, chỉ cần $n \geq 10$ thì $f(n) = 2n + 10 \leq 3n = c \times n$
- $2n^2 + 10$ thì không phải là $O(n)$ nữa, mà sẽ là $O(n^2)$ (chọn $c = 3$ và $n \geq 4$)

Phân tích độ phức tạp thuật toán

- Bằng lý thuyết tính toán ,xấp xỉ tiệm cận.

- Bằng thời gian chạy thực nghiệm

  • Sử dụng hàm clock( ) để đo thời gian chạy chương trình

```
begin = clock();
bubbleSort(b, n);
timeUsed = (double)(clock() - begin) / CLOCKS_PER_SEC;
```

- Khi phân tích độ phức tạp của thuật toán người ta không quan tâm tới cấu hình máy, ngôn ngữ lập trình nó chỉ dựa vào kích thước đầu vào được xác định.

Phân tích độ phức tạp thuật toán

```
int a = 5;
int b = 7;
int c = 4;
int d = a + b + c + 153;
```

O(1)

```
int i = 0;
while (i < n) {
        i++;
}
```

O(n)

```
for (int i = 1; i <= 5 * n + 17; i++) {
}
```

O(n)

```
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
        }
}
```

O(n.m)

2.1 Selection Sort

Giải thuật sắp xếp lựa chọn

- Bước 1: Thiết lập Min = 1 là vị trí đầu tiên của dãy
- Bước 2: Tìm kiếm phần tử nhỏ nhất trong danh sách
- Bước 3: Tráo đổi giá trị tại vị trí Min
- Bước 4: Tăng Min để trỏ tới vị trí tiếp theo
- Bước 5: Lặp lại từ bước 2 cho đến khi danh sách được sắp xế

No.	Min	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
		32	51	27	83	66	11	45	75
1	11	11	51	27	83	66	32	45	75
2	27	11	27	51	83	66	32	45	75
3	32	11	27	32	83	66	51	45	75
4	45	11	27	32	45	66	51	83	75
5	51	11	27	32	45	51	66	83	75
6	66	11	27	32	45	51	66	83	75
7	75	11	27	32	45	51	66	75	83

Chiến thuật: Chọn số nhỏ nhất trong dãy chưa được sắp xếp và đổi chỗ với số đang chiếm vị trí đầu tiên của dãy này

2.2 Insertion Sort

Giải thuật sắp xếp chèn

- Bước 1: Xét A[1] là dãy con ban đầu đã được sắp xếp
- Bước 2: Xét A[2], nếu A[2] < A[1] chèn vào trước A[1], còn lại thì giữ nguyên A[2] tại chỗ
- Bước 3: Xét A[1], A[2] là dãy con được sắp xếp
- Bước 4: Xét A[3], so sánh với A[1], A[2] và tìm vị trí chèn
- Bước 5: Lặp lại với A[4], … đến khi dãy được sắp xếp hết

No.	Số so sánh	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
		32	51	27	83	66	11	45	75
1	51	32	51	27	83	66	11	45	75
2	27	27	32	51	83	66	11	45	75
3	83	27	32	51	83	66	11	45	75
4	66	27	32	51	66	83	11	45	75
5	11	11	27	32	51	66	83	45	75
6	45	11	27	32	45	51	66	83	75
7	75	11	27	32	45	51	66	75	83

2.3 Bubble sort

Nguyên tắc

– Duyệt bảng khoá (danh sách khoá) từ đáy lên đỉnh

– Dọc đường nếu thứ tự 2 khoá liền kế không đúng => đổi chỗ

• Nhận xét

– Khoá nhỏ sẽ nổi dần lên sau mỗi lần duyệt => “nổi bọt”

– Sau một vài lần (không cần chạy n bước), danh sách khoá đã có thể được sắp xếp => Có thể cải tiến thuật toán, dùng 1 biến lưu trạng thái, nếu không còn gì thay đổi (không cần đổi chỗ) => ngừng

		1	2	3	4	5	6	7
A[1]	32	11	11	11	11	11	11	11
A[2]	51	32	27	27	27	27	27	27
A[3]	27	51	32	32	32	32	32	32
A[4]	83	27	51	45	45	45	45	45
A[5]	66	83	45	51	51	51	51	51
A[6]	11	66	83	66	66	66	66	66
A[7]	45	45	66	83	75	75	75	75
A[8]	75	75	75	75	83	83	83	83

Chiến thuật: Dựa trên việc so sánh cặp phần tử liên kề nhau và tráo đổi vị trí nếu chúng không theo thứ tự

2.4 Quick Sort

Hiệu năng thực thi tốt hơn

– Chia để trị

– Giải thuật sắp xếp đệ quy

• Phần tử được chọn là bất kỳ được gọi là “chốt” (pivot)

• Mọi phần tử nhỏ hơn chốt sẽ được đẩy lên phía trước chốt

• Hai mảng con:

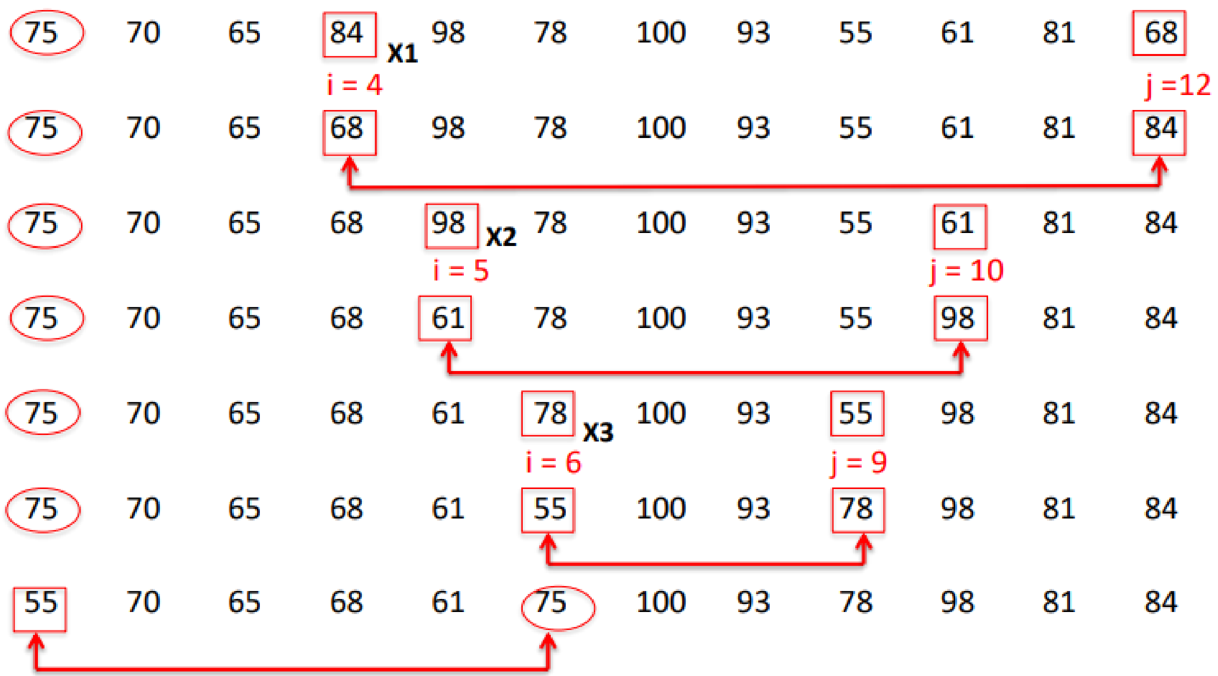
– Mảng con nhỏ hơn chốt ở phía trước chốt

– Mảng con lớn hơn chốt ở phía sau chốt

• Chiến thuật tương tự với từng mảng con, đến khi mảng con chỉ còn một phần tử

2.4 Quick Sort

- Xét mảng A có các phần tử sau:
- 75, 70, 65, 84, 98, 78, 100, 93, 55, 61, 81, 68
- Bước 1: Giả sử chọn 75 làm chốt
- Bước 2: Thực hiện phép tìm kiếm các số nhỏ hơn 75
- và lớn hơn 75
- Bước 3: Thu được 2 mảng con sau
- 70, 65, 55, 61, 68 và 84, 98, 100, 93, 81
- Bước 4: Quá trình sắp xếp tương tự với 2 mảng con
- trên



2.4 Quick Sort

Phân đoạn

```
Partition(A, first,last){
        if (first>=last) return;
        c=A[first];//phần tử chốt
        i=first+1,j=last;
        while (i<=j){
                while (A[i]<=c && i<=j) i++;
                while (A[j]>c && i<=j) j--;
                if (i<j) swap(A[i],A[j]);
                }
        swap(A[first],A[j]);
        Partition(A, first,j-1);
        Partition(A, j+1,last);
}
```

Sắp xếp

```
Procedure QuickSort(A, N){
Partition(A,0,N-1);
}
```

2.4 Quick Sort

- Độ phức tạp trong trường hợp xấu nhất là

T(n) = O(N^2)

- Độ phức tạp trong trường hợp tốt nhất là – Khi các lần chia sẽ tạo ra mảng con = ½ mảng cha

T(n) = O(n$\log n$)

- Độ phức tạp trung bình

T(n) = O(nlogn)

- Khi n lớn thì Quick_sort có hiệu năng tốt hơn đa số các

phương pháp còn lại

2.5 Heap sort

- Cấu trúc đống
- Phép tạo đống
- Sắp xếp kiểu vun đống (Heap – sort)

Cấu trúc đống

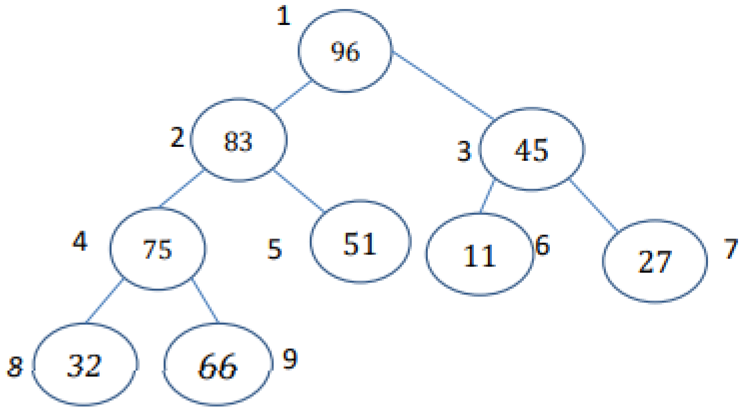
• Đống là một cây nhị phân mà mỗi nút gắn với một số sao cho số ở nút cha bao giờ cũng lớn hơn số ở nút con

• Ví dụ: dùng cây nhị phân hoàn chỉnh

– Số ứng với gốc của đống chính là số lớn nhất

– Biểu diễn trong máy dưới dạng vector như sau

96	83	45	75	51	11	27	32	66
1	2	3	4	5	6	7	8	9

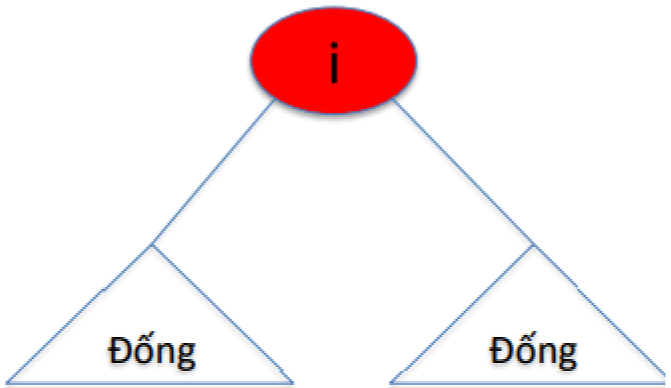


Giải thuật tạo đống

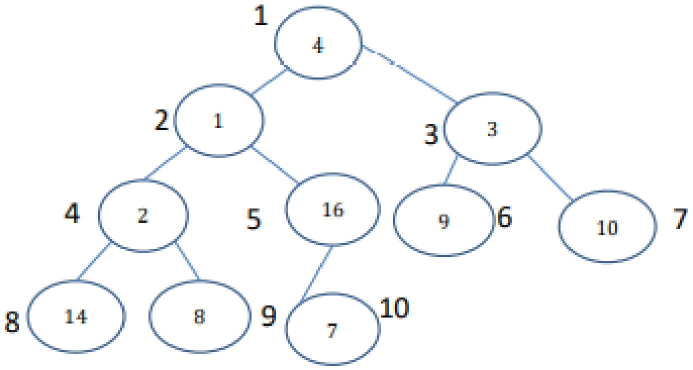
```
void heapRebuild(int a[], int pos, int n)
{

    while (2 * pos + 1 < n)
    {
        int j = 2 * pos + 1;
        if (j < n - 1)
            if (a[j] < a[j + 1])
                j = j + 1;
        if (a[pos] >= a[j])
            return;
        HoanVi(a[pos], a[j]);
        pos = j;
    }
}
void heapConstruct(int a[], int n)
{
    for (int i = (n - 1) / 2; i >= 0; i--)
        heapRebuild(a, i, n);
}
```

- Có n nút biểu diễn bởi vector A, lệnh sau đây thực hiện tạo cây nhị phân hoàn chỉnh thành đống

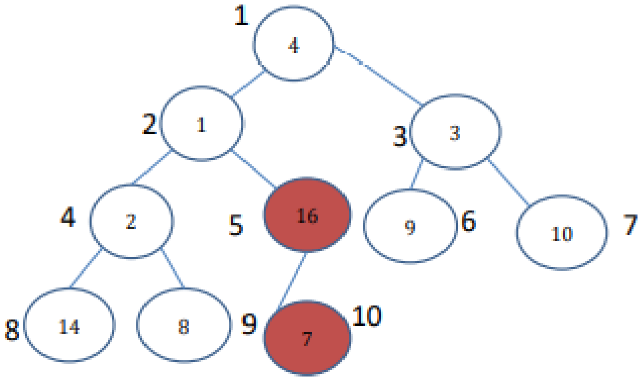
  For i = n/2 down to 1 Call heapRebuild (a,i,n)
- Hàm tạo đống: heapRebuild (a,i,n)



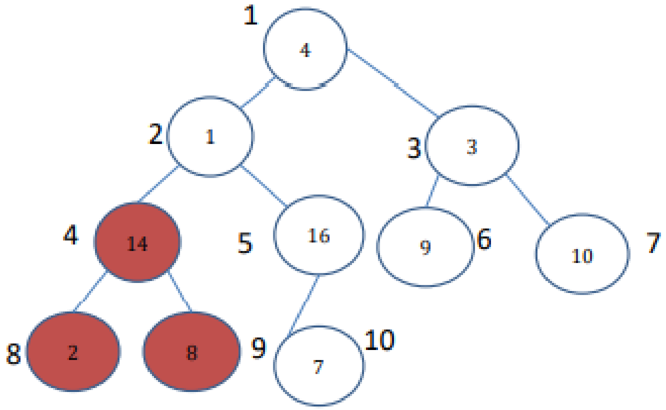
Mô tả các bước vun đống



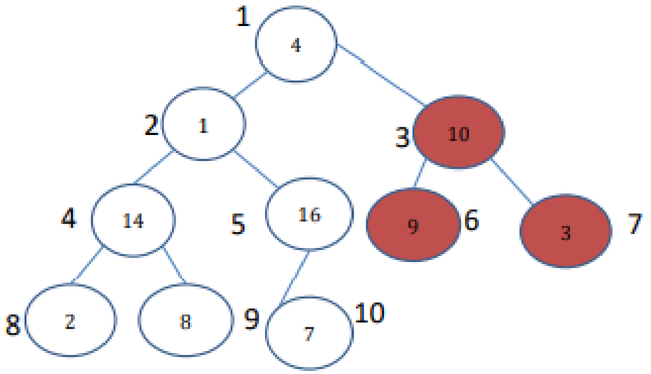
Tạo đống ban đầu



Thực hiện Adjust(5,10)

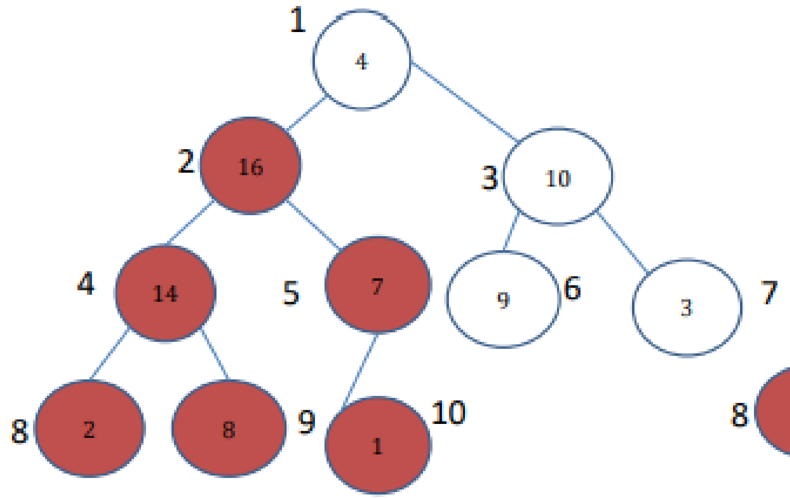


Thực hiện Adjust(4,10)

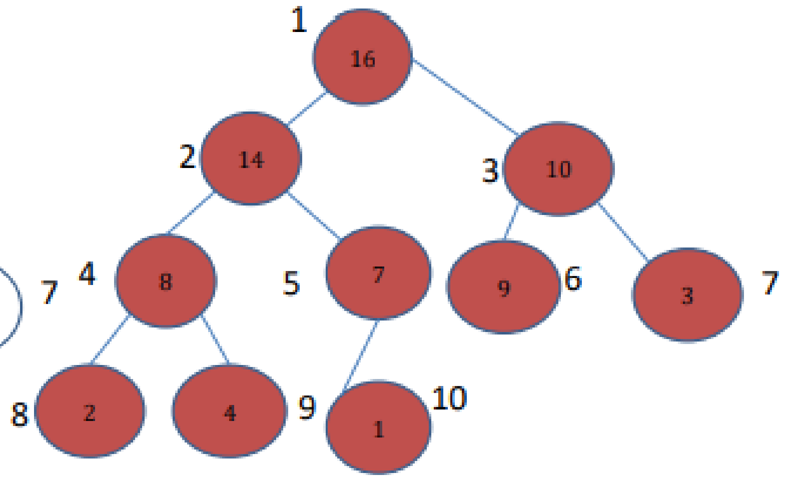


Thực hiện Adjust(3,10)

Mô tả các bước vun đống



Thực hiện Adjust(2,10)



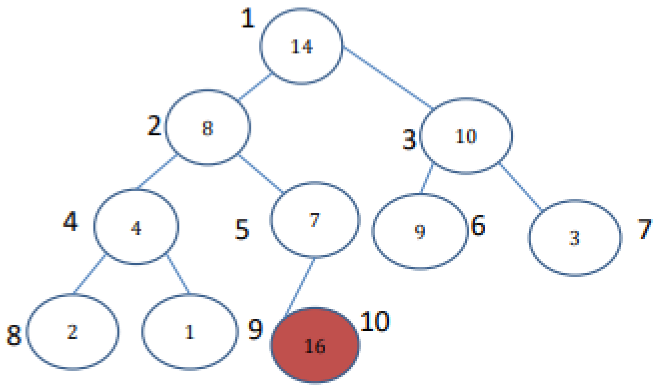
Thực hiện Adjust(1,10)

Lúc này cây đã là đống và biểu diễn trong máy là vector A như sau

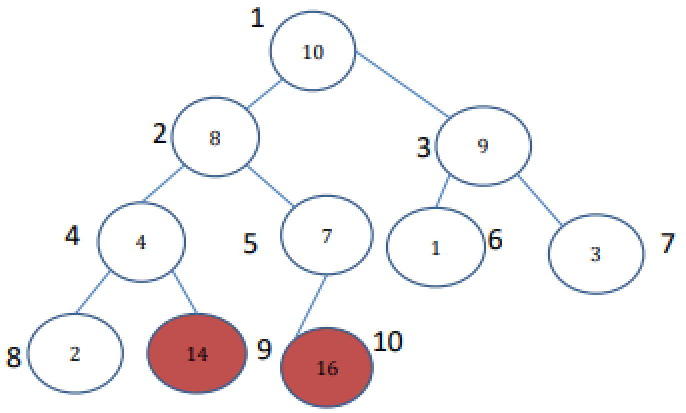
16	14	10	8	7	9	3	2	4	1

Giải thuật sắp xếp vun đống

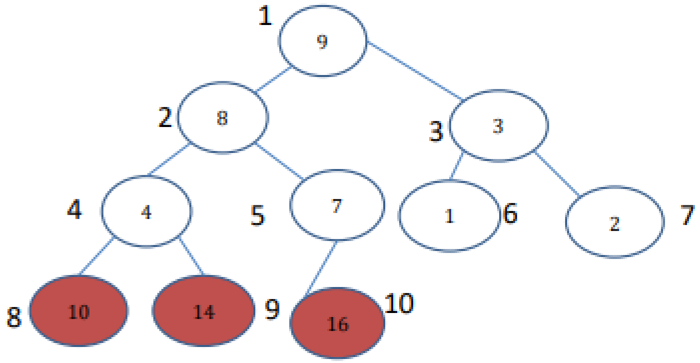
```
void heapSort(int a[], int n)
{
    heapConstruct(a, n);
    int r = n - 1;
    while (r > 0)
    {
        HoanVi(a[0], a[r]);
        heapRebuild(a, 0, r);
        r--;
    }
}
```



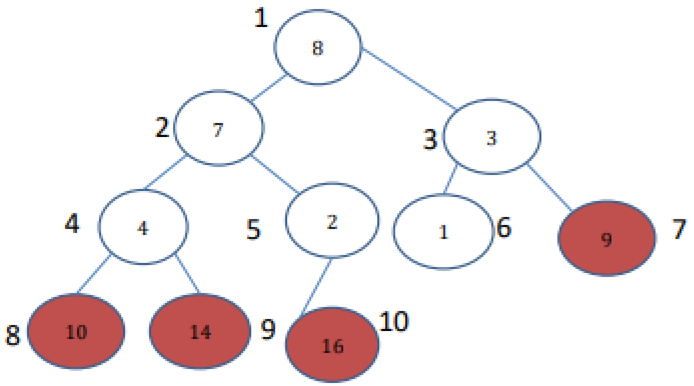
Đổi lần 1: giữa A[1] và A[10], vun đống cho cây với 9 nút còn lại, 16 đã vào đúng vị trí



Đổi lần 2: giữa A[1] và A[9], vun đống cho cây với 8 nút còn lại, 14, 16 vào đúng vị trí

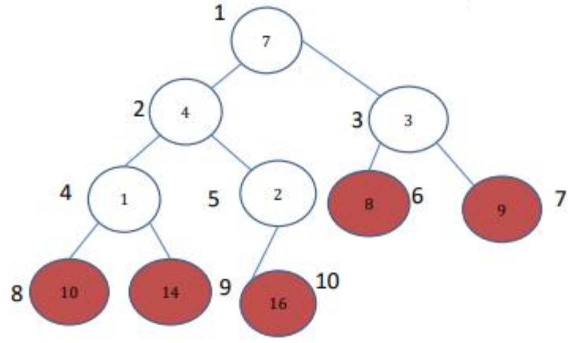


Đổi lần 3: giữa A[1] và A[8], vun đống cho cây với 7 nút còn lại, 10,14, 16 vào đúng vị trí

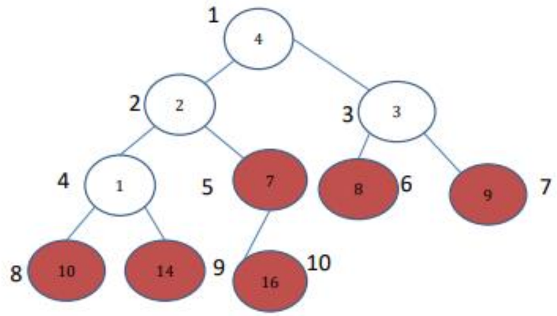


Đổi lần 4: giữa A[1] và A[7], vun đống cho cây với 6 nút còn lại

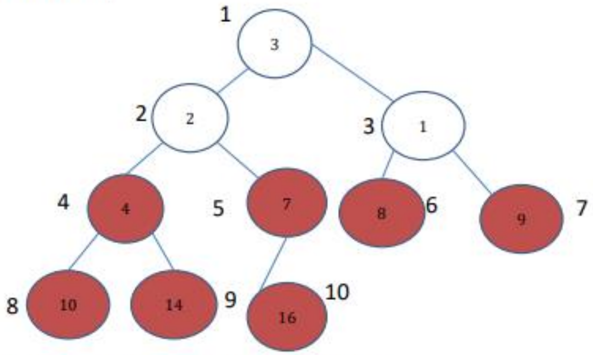
Giải thuật sắp xếp vun đống



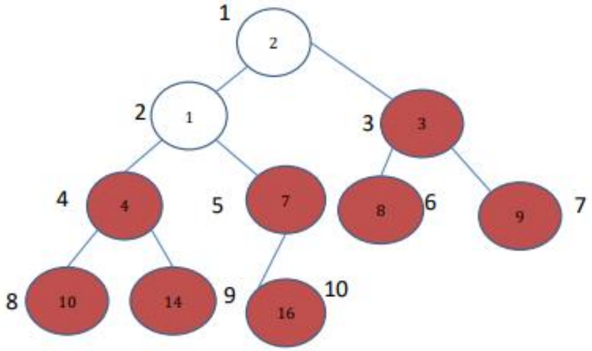
Đổi lần 5: giữa A[1] và A[6], vun đống cho cây với 5 nút còn lại



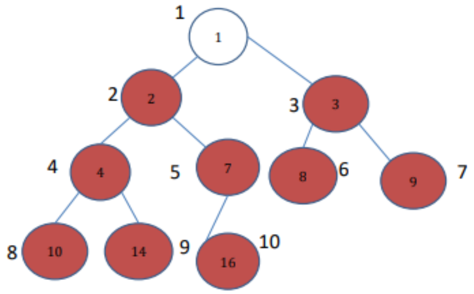
Đổi lần 6: giữa A[1] và A[5], vun đống cho cây với 4 nút còn lại



Đổi lần 7: giữa A[1] và A[4], vun đống cho cây với 3 nút còn lại



Đổi lần 8: giữa A[1] và A[3], vun đống cho cây với 2 nút còn lại



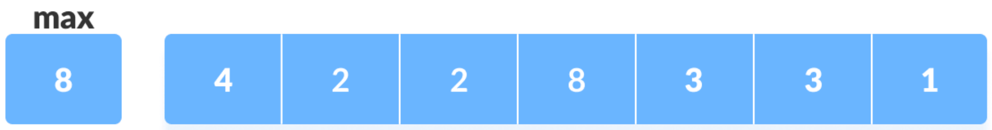
1	2	3	4	7	8	9	10	14	16

Đổi lần 9: giữa A[1] và A[2], đống chỉ còn 1 nút.
Dãy A đã được sắp xếp

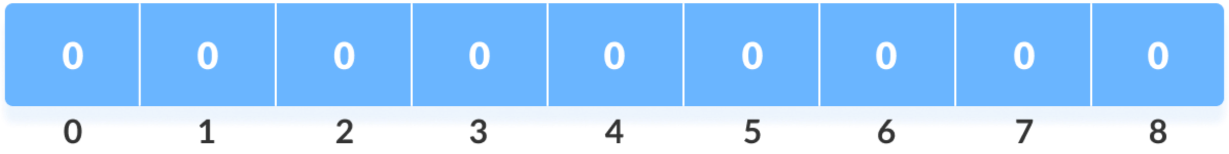
T(n) = O(nlogn)

2.6 Couting sort

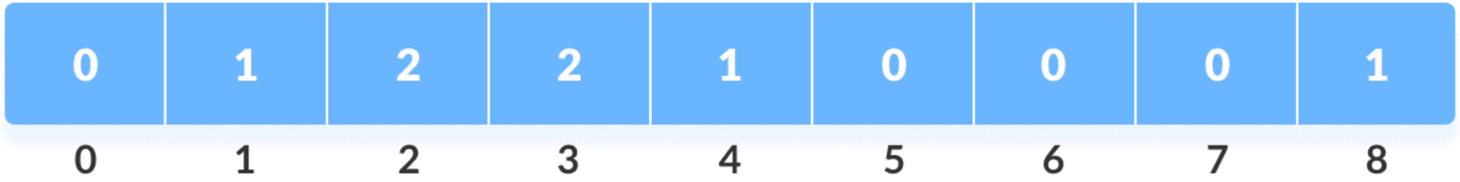
1. Tìm phần tử lớn nhất (gọi là max) từ mảng đã cho.



2. Khởi tạo một mảng có độ dài max+1với tất cả các phần tử bằng 0. Mảng này được sử dụng để lưu trữ số phần tử trong mảng.

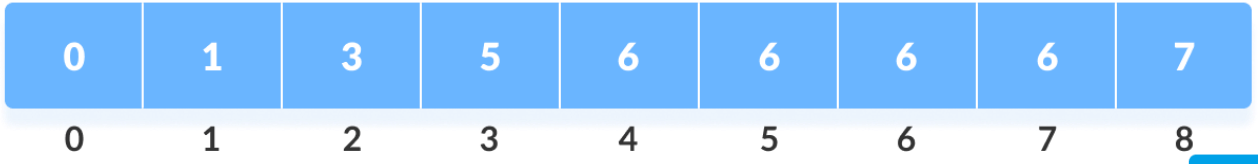


3. Lưu trữ số lượng của từng phần tử tại chỉ mục tương ứng của chúng trong count mảng



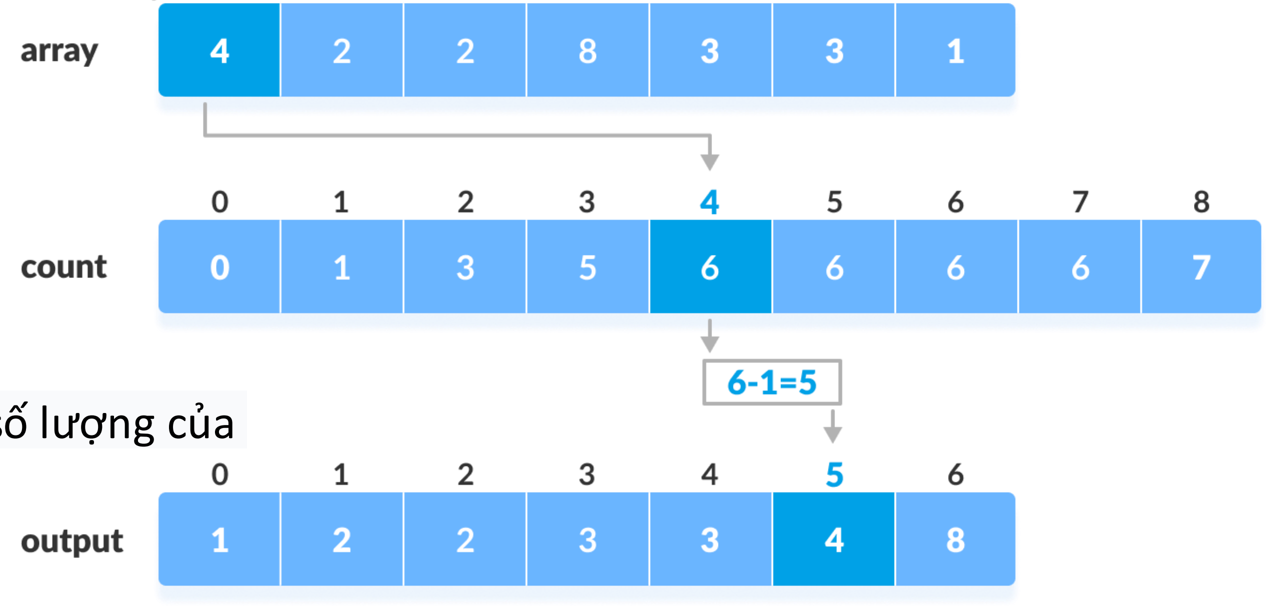
2.6 Couting sort

4. Lưu trữ tổng tích lũy của các phần tử của mảng đếm. Nó giúp đặt các phần tử vào đúng chỉ mục của mảng đã sắp xếp.



5. Tìm chỉ số của từng phần tử của mảng ban đầu trong mảng đếm. Điều này cho số lượng tích lũy. Đặt phần tử tại chỉ mục được tính toán như trong hình .

6. Sau khi đặt từng phần tử vào đúng vị trí, giảm số lượng của nó đi một.



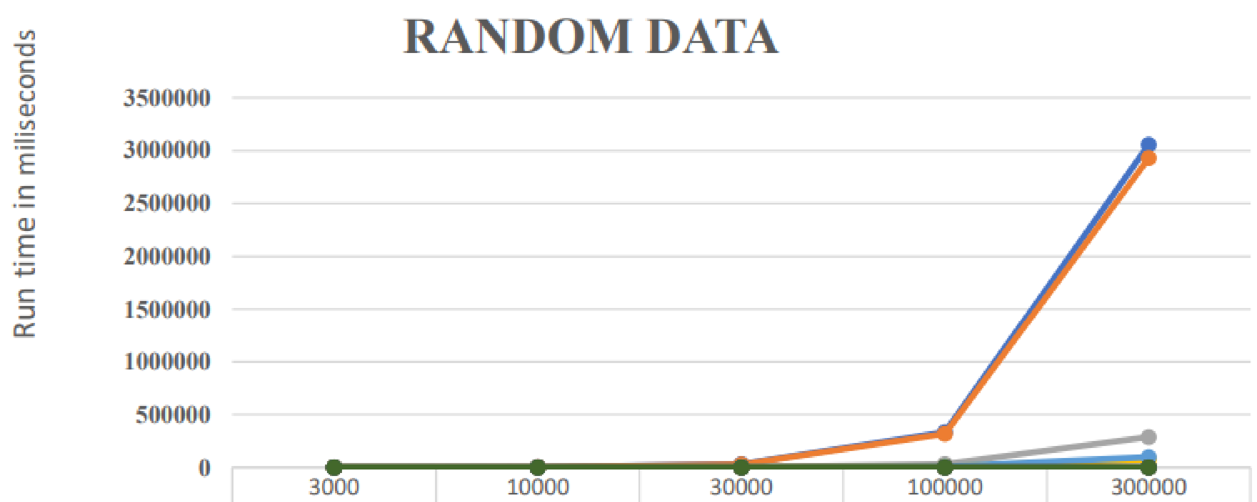
```
void countingSort(int a[], int n)
{
    int max = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];                    O(n)

    int *count = new int[max + 1];
    for (int i = 0; i <= max; i++)         O(max)
        count[i] = 0;

    for (int i = 0; i < n; i++)
        count[a[i]]++;                     O(n)

    for (int i = 1; i <= max; i++)
        count[i] += count[i - 1];          O(max)

    int *temp = new int[n];
    for (int i = 0; i < n; i++)
    {
        temp[count[a[i]] - 1] = a[i];      O(n)
        count[a[i]]--;
    }

    for (int i = 0; i < n; i++)
        a[i] = temp[i];
    delete[] count;
    delete[] temp;
```

- **Độ phức tạp của trường hợp xấu nhất:** O(n+k)
- **Độ phức tạp của trường hợp tốt nhất:** O(n+k)
- **Độ phức tạp trường hợp trung bình:** O(n+k)

Space Complexity O(max)

Bảng đánh giá

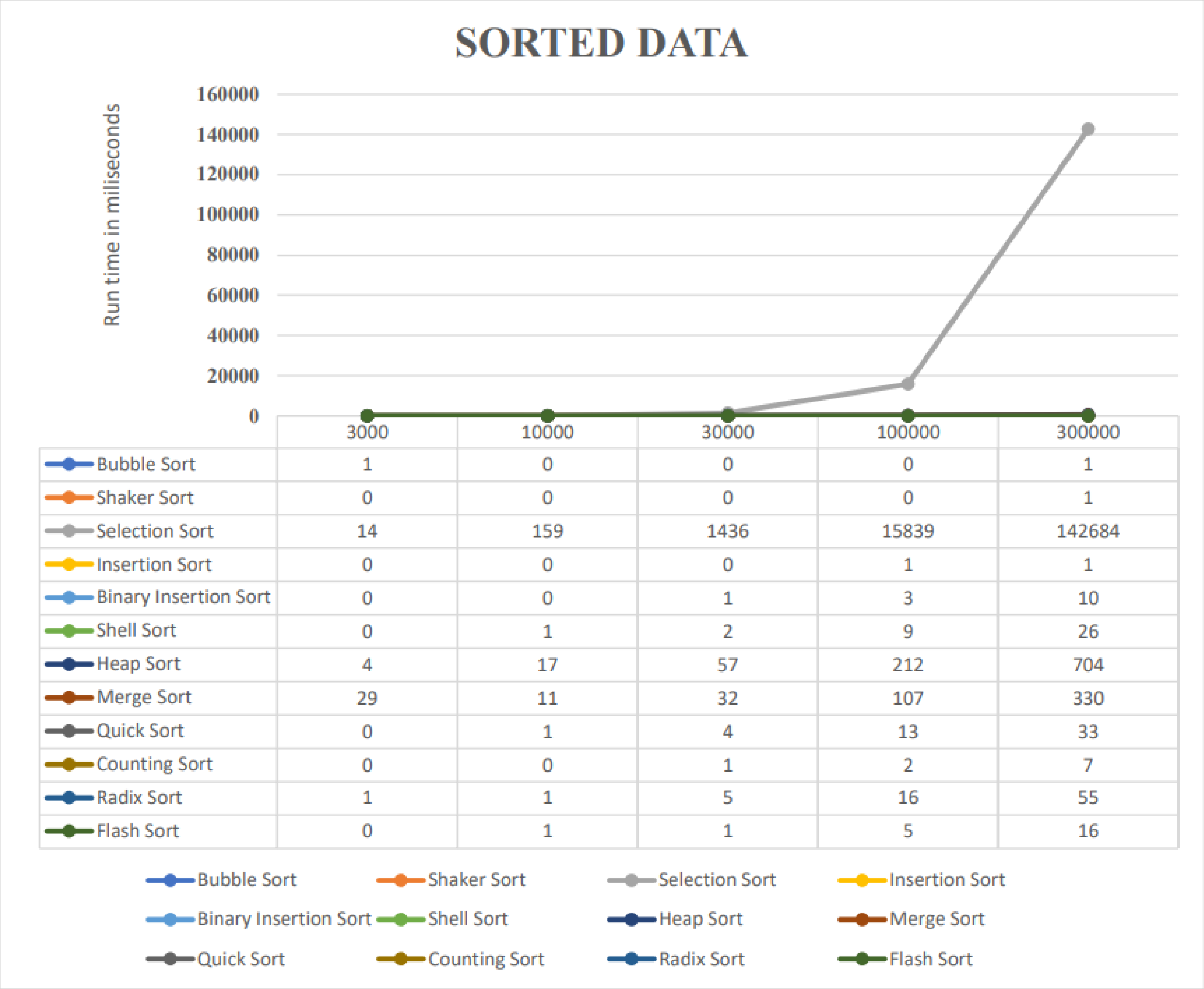
		Độ phức tạp			
STT	Thuật toán	Tốt nhất	Trung bình	Xấu nhất	Bộ nhớ
1	Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
2	Shaker Sort	O(n)	O(n^2)	O(n^2)	O(1)
3	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
4	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
5	Binary Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
6	Shell Sort	O(nlogn)	depends on gap sequence	O(n^2)	O(1)
7	Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
8	Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
9	Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(logn)
10	Counting Sort	O(n+k)	O(n + k)	O(n + k)	O(n + k)
11	Radix Sort	O(kn)	O(nk)	O(nk)	O(n + k)
12	Flash Sort	O(n)	O(n + r)	O(n^2)	O(m)



	3000	10000	30000	100000	300000
Bubble Sort	370	3404	30175	334109	3056580
Shaker Sort	287	3436	28946	316954	2928490
Selection Sort	35	322	2989	31680	288024
Insertion Sort	8	84	752	8210	74336
Binary Insertion Sort	5	46	423	8549	97915
Shell Sort	0	4	11	37	122
Heap Sort	5	17	56	220	711
Merge Sort	3	12	36	118	365
Quick Sort	0	6	6	22	63
Counting Sort	1	1	1	2	8
Radix Sort	0	1	5	15	47
Flash Sort	0	0	2	6	22

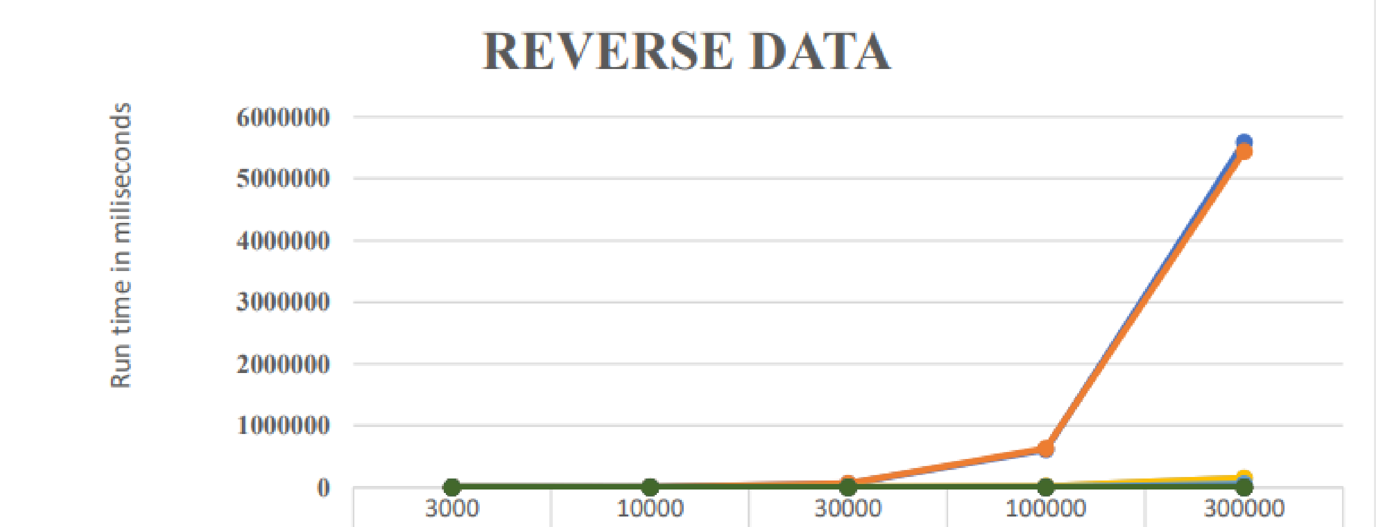
Bubble Sort, Shaker Sort, Selection Sort, Insertion Sort, Binary Insertion Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, Flash Sort

Bảng thống kê với dữ liệu đầu vào ngẫu nhiên



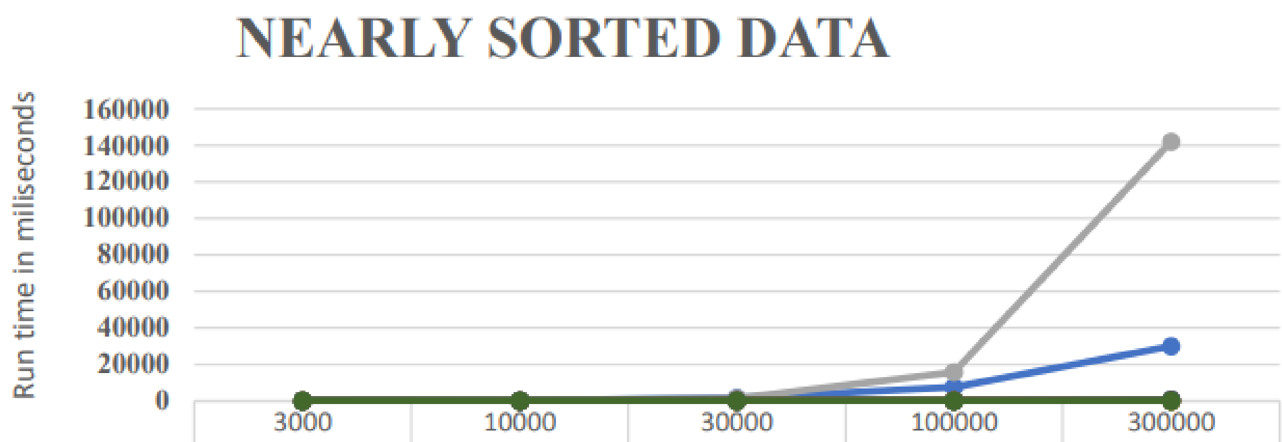
	3000	10000	30000	100000	300000
Bubble Sort	1	0	0	0	1
Shaker Sort	0	0	0	0	1
Selection Sort	14	159	1436	15839	142684
Insertion Sort	0	0	0	1	1
Binary Insertion Sort	0	0	1	3	10
Shell Sort	0	1	2	9	26
Heap Sort	4	17	57	212	704
Merge Sort	29	11	32	107	330
Quick Sort	0	1	4	13	33
Counting Sort	0	0	1	2	7
Radix Sort	1	1	5	16	55
Flash Sort	0	1	1	5	16

Bảng thống kê với dữ liệu đầu vào có thứ tự tăng dần



	3000	10000	30000	100000	300000
Bubble Sort	543	6072	56516	613504	5587380
Shaker Sort	548	6062	58897	630679	5439340
Selection Sort	15	172	1495	17169	148868
Insertion Sort	15	164	1480	16725	148053
Binary Insertion Sort	6	61	542	6127	54483
Shell Sort	0	0	3	11	41
Heap Sort	5	16	53	230	652
Merge Sort	3	10	32	132	326
Quick Sort	0	1	3	11	39
Counting Sort	0	0	1	3	8
Radix Sort	0	1	5	17	57
Flash Sort	0	0	2	4	14

Bảng thống kê với dữ liệu đầu vào có thứ tự giảm dần



	3000	10000	30000	100000	300000
Bubble Sort	17	173	1527	7489	29960
Shaker Sort	2	9	18	30	24
Selection Sort	15	158	1426	15767	141974
Insertion Sort	0	0	1	1	2
Binary Insertion Sort	0	1	1	4	10
Shell Sort	1	1	3	10	29
Heap Sort	4	17	57	213	696
Merge Sort	3	10	31	106	325
Quick Sort	0	1	3	12	35
Counting Sort	0	0	1	2	8
Radix Sort	1	1	5	15	54
Flash Sort	0	1	2	6	15

Bảng thống kê với dữ liệu đầu vào gần như có thứ tự tăng dần

3. Tìm kiếm trên cấu trúc mảng

Bài toán tìm kiếm trên cấu trúc mảng

• Tìm kiếm một phần tử theo một tiêu chí nào đó
• Tìm kiếm “trả về được ” khi có hoặc “trả về không” khi không có phần tử nào
– Tìm kiếm tuần tự
– Tìm kiếm nhị phân
• Ví dụ: mảng A gồm n phần tử A[1], A[2]…A[n]
Cho số X, tìm xem có giá trị nào trong A bằng X hay không?

3. Tìm kiếm trên cấu trúc mảng

1. Tìm kiếm tuần tự

• Duyệt tất cả các phần tử trong mảng A

– Nếu có phần tử nào bằng X thì ghi nhận lại chỉ số của phần tử đó,

– Nếu không có thì ghi nhận bằng 0.

2. Tìm kiếm nhị phân

• So sánh X với phần tử A[k] ở giữa mảng

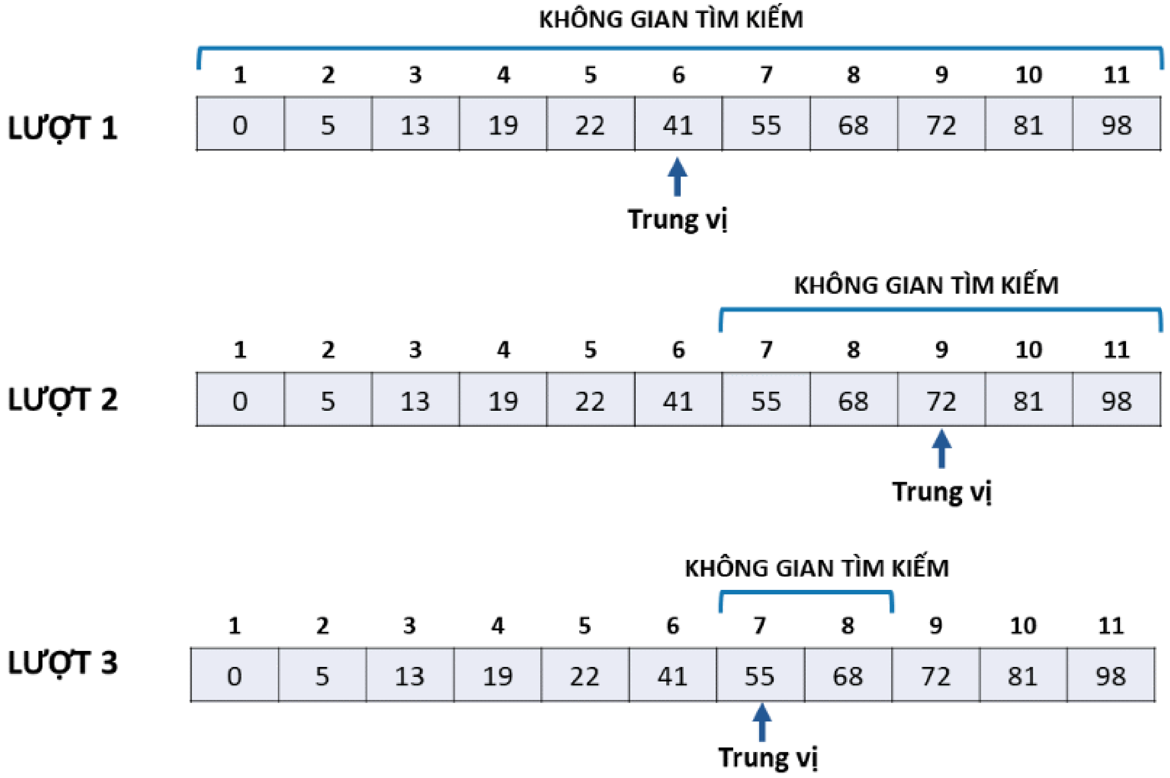
– Nếu X< A[k] tìm kiếm với nửa đầu của mảng A

– Nếu X> A[k] tìm kiếm với nửa cuối của mảng A

– Nếu X = A[k] tìm kiếm được thỏa

3.1 Tìm kiếm nhị phân

Cho A=[0,5,13,19,2,41,55,68,72,81,98] và x=55, thuật toán sẽ diễn ra như hình dưới:



3.1 Tìm kiếm nhị phân

```c
int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}
```

```c
int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

input

```
Element is found at index 1
```

Độ phức tạp thời gian

- **Độ phức tạp của trường hợp tốt nhất** :O(1)
- **Độ phức tạp trường hợp trung bình** :O(log n)
- **Độ phức tạp của trường hợp xấu nhất** :O(log n)

Độ phức tạp của không gian

- **Độ phức tạp không gian của tìm kiếm nhị phân là** O(1).