Các thuật toán sắp xếp

Các thuật toán sắp xếp gồm hai loại: sắp xếp nội (internal sorting), đưa toàn bộ dữ liệu vào bộ nhớ trong. Sắp xếp nội có kích thước dữ liệu không lớn và thời gian thực hiện thuật toán nhanh. Loại thứ hai là sắp xếp ngoại (external sorting), chỉ có một phần dữ liệu được đưa vào bộ nhớ trong, phần còn lại lưu ở bộ nhớ ngoài. Sắp xếp ngoại có kích thước dữ liệu rất lớn và thời gian thực hiện chậm.

Internal memory – bộ nhớ trong, chính xác là RAM. Sẽ mất dữ liệu khi tắt máy. Truy xuất dữ liệu ngẫu nhiên.

External memory – bộ nhớ ngoài, ngoài RAM hoặc Register. Điển hình là HDD hoặc SSD. Không mất dữ liệu khi tắt máy. Truy xuất dữ liệu trực tiếp phụ thuộc vào vị trí vật lý của dữ liệu.

Inplace : Không dùng bộ nhớ trung gian.

Stable : Một thuật toán được gọi là stable khi chúng giữ nguyên thứ tự của các phần tử tương tự tính chất với nhau. Chẳng hạn như ta có dãy

$$1, 7, 4, 5a, 5b, 5c, 7, 6$$

Sau khi sắp xếp xong, thuật toán Stable sẽ không đảo thứ tự giữa 5a, 5b và 5c. Còn các thuật toán Unstable có thể đảo thành 5a, 5c, 5b hoặc 5c, 5b, 5a gì đấy.

Một ví dụ khác là:

$$1, 9, 82, 2, 19, 27$$

Một thuật toán Stable, sau khi sắp xếp dãy trên xong, thứ tự giữa 82 và 2 là không đổi, do chúng có cùng tính chất là giống nhau hàng đơn vị (82 và 02).

Ngoại trừ Radix Sort thì còn lại đều là **Comparison-based Sorting Algorithm**. Tất cả các thuật toán Comparision-based mang tính tổng quát (General Purpose) thì lower bound là O(nlog(n)). Còn Radix Sort, mang tính đặc dụng hơn (Specific Purpose), thì có lower bound là O(n).

Chúng ta phải xem xét các thuật toán dựa trên một ngữ cảnh nhất định.

Internal Sorting

Có bốn loại đó là

- Đổi chỗ (Exchange Sort) gồm Bubble Sort, Shaker Sort, Interchange Sort, Quick Sort,..
- Chèn (Insertion Sort) gồm Simple Insertion Sort, Binary Insertion Sort, Shell Sort,...
- Trộn (Merge Sort) gồm Merge Sort, Natural Merge Sort,...
- Lựa chọn (Selection Sort) gồm Simple Selection Sort, Heap Sort,...

Trong đó Shell Sort, Heap Sort, Quick Sort, Merge Sort và Radix Sort là các thuật toán phức tạp nhưng chi phí thấp. Các thuật toán còn lại đơn giản nhưng chi phí cao.

Interchange Sort

Nghịch thế là một cặp giá trị (a_i, a_j) khi a_i và a_j không thỏa điều kiện sắp thứ tự. Ví dụ nếu mảng một chiều có các phần tử tăng dần mà có một cặp (ai, aj) nào đó giảm dần thì cặp đó được gọi là **nghịch thế**.

Ý tưởng

Thuật toán Interchange Sort sẽ duyệt qua tất cả các cặp giá trị trong mảng và hoán vị hai giá trị trong một cặp nếu cặp giá trị đó là **nghịch thế**.

Đầu vào – Đầu ra

- Input: Mảng A gồm n phần tử chưa sắp xếp.
- Output: Mảng A đã sắp xếp.

Kể từ thuật toán này trở đi, đầu vào và đầu ra của hầu hết các thuật toán sắp xếp khác đều như nhau.

Độ phức tạp thuật toán

- **Best case** : *Số lần so sánh*: n(n-1)/2. *Số lần hoán vị*: 0.
- **Worst case** : *Số lần so sánh*: n(n-1)/2. *Số lần hoán vị*: n(n-1)/2.
- **Average Case** : Độ phức tạp đa thức $O(n^2)$.

Giải thuật mẫu

```
void interchangeSort(int* a,int n)
{
    // Do xét cặp nên chỉ xét đến phần tử kế cuối
    for (int i = 0; i > n - 1; i++)
    {
        // So sánh với các phần tử còn lại khác chính nó_
        for (int j = i + 1; j < n;j++){
            //Giả sử xếp tăng dần
            if(a[j] > a[i])
            {
                a[i] = a[j] + a[i];
                a[j] = a[i] - a[j];
                a[i] = a[i] - a[j];
            }
        }
    }
}
```

Selection Sort

Ý tưởng

Chọn phần tử nhỏ nhất (cực trị) cho các vị trí từ 0 đến n – 1.

Lần chọn 0: Chọn phần tử nhỏ nhất (a[min]) từ 0 đến n – 1. Đổi chỗ hai nút tại vị trí min và 0.

Lần chọn 1: Chọn phần tử nhỏ nhất (a[min]) từ 1 đến n – 1. Đổi chỗ hai nút tại vị trí min và 1.

Lần chọn i: Chọn phần tử nhỏ nhất (a[min]) từ i đến n – 1. Đổi chỗ hai nút tại vị trí min và i.

Lần chọn cuối: n – 2.

Điểm thú vị của selection là có thể sort trong khoảng từ 0 đến k (k < n). Ví dụ khi tuyển sinh, ta chỉ tuyển 500 học sinh đầu trong 1000 học sinh. Chúng ta sắp xếp theo tên của các học sinh, khi sắp xếp đến số lượng 500 thì ngừng, không cần duyệt qua toàn bộ dữ liệu đầu vào. Vậy Selection Sort nó có thể ngừng sort tại vị trí mong muốn nào đó trong quá trình sort, giúp tiết kiệm chi phí.

Dễ hiểu, dễ cài đặt, dùng trong **(prototype?)**. Tức là khi mình build một hàm lớn, cần sắp xếp dữ liệu, chúng ta chọn Selection Sort để làm điều đó. Sau này, khi chúng ta cần tối ưu hóa thời gian thực hiện thì thay thế bằng các thuật toán khác mà không làm ảnh hưởng đến kết quả xử lý.

Độ phức tạp thuật toán

- **Best case** : Ở lượt thứ i, lúc nào cũng cần n – i – 1 lượt so sánh để xác định phần tử nhỏ nhất hiện hành.

Nên *số lần so sánh*:

$$\sum_{i=0}^{n-2} n-i-1 = (n-1)+(n-2)+\ldots+(n-i-1)+\ldots+1 = n(n-1)/2.$$

Số lần hoán vị: 0.

Độ phức tạp là: $O(n)$.

- **Worst case** : Số lần so sánh: n(n – 1)/2. Số lần hoán vị: n – 1.
- **Average case** : $O(n^2)$.

Giải thuật mẫu

```
void selectionSort(int *a, int n)
{
    // Phần tử kế cuối đã tự sắp xếp_
    for(int i = 0;i < n - 1; i++)
    {
        int min = i;
        for(int j = i + 1;j < n; j++)
        {
            if(a[j] < a[min])

                min = j;
        }
        swap(a[i], a[min]);
    }
}
```

Bubble Sort

Ý tưởng

Xuất phát từ đầu dãy hoặc cuối dãy, đổi chỗ các cặp phần tử liền kề để đưa phần tử nhỏ hơn trong cặp phần tử đó về đúng đầu dãy hiện hành. Sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i thì vị trí đầu dãy là i.

Cần phân biệt với Interchange Sort khi thuật toán Bubble Sort không so sánh tất cả các cặp tồn tại mà chỉ so sánh các cặp nghịch thế liền kề với nhau.

Độ phức tạp thuật toán

- **Best case** : *Số lần so sánh* là n(n – 1)/2, *số lần hoán vị* là 0. Độ phức tạp: O(n) (mảng đã sắp xếp).
- **Worst case** : *Số lần so sánh* là n(n – 1)/2, *số lần hoán vị* là n(n – 1)/2. Độ phức tạp: O(n2)
- **Average case** : O(n2).

Giải thuật mẫu

```
void bubbleSort(int *a,int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        bool isSorted = true;// Tạo ra một cờ lệnh
        //?Do chúng ta so sánh hai phần tử j và j + 1,nên: j + 1 < n
        //?Sau mỗi vòng lặp thì giảm đi một phần tử nên trừ đi i: j + 1 < n - i
        //?Chuyển vế đổi dấu ta có điều kiện là j < n - i - 1

        for(int j = 0; j < n - i - 1; j++){
            if(a[j] < a[j + 1])
            {
                swap(a[j], a[j + 1]);
                isSorted = false;
            }
        }

        //Nếu như không có sự thay đổi nào thì dừng sắp xếp.
        if(isSorted) return;
    }
}
```

Shaker Sort (Cocktail Sort)

Ý tưởng

Trong mỗi lần sắp xếp, duyệt mảng theo 2 lượt từ hai phía khác nhau:

Lượt đi: đẩy phần tử nhỏ về đầu mảng.

Lượt về: đẩy phần tử lớn về cuối mảng.

Ghi nhận lại các đoạn đã sắp xếp nhằm tiết kiệm các phép so sánh thừa.

Shaker Sort là một dạng nâng cao của Bubble Sort nên nó có thể nhận diện được mảng đã sắp xếp. Đồng thời Shaker Sort sẽ tối ưu hơn Bubble Sort trong trường hợp dãy đã gần như có thứ tự. Ví dụ {2,3,4,5,1} thì Shaker Sort cần 2 lần đi và về, Bubble Sort cần 4 lần duyệt. Tuy nhiên nếu như Bubble Sort có cờ lệnh thì cũng sẽ còn 2 lần lặp (1 lần sắp xếp và 1 lần để biết mảng đã có thứ tự).

Độ phức tạp thuật toán

- **Best case** : $O(n)$ khi mảng đã hoàn toàn sắp xếp.
- **Worst case** : $O(n^2)$.
- **Average case** : $O(n^2)$.

Giải thuật mẫu

```
void shakerSort(int *a,int n)
{
    int left = 0, right = n - 1,k = n - 1;
    while(left < right)
    {
        //Lượt đi index giảm dần, đổi chỗ các cặp nghịch thế liền kề
        for(int i = right; i > left; i--)
        {
            if(a[i] <  a[i - 1]){
                swap(a[i], a[i - 1]);
                k = i;//Lưu lại vị trí có hoán vị
            }
        }

        //Loại bỏ các đoạn đã sắp xếp
        left = k;

        //Lượt về index tăng dần
        for(int i = left;i < right;i++)
        {
            if(a[i] > a[i + 1]){
                swap(a[i],a[i + 1]);
                k = i;
            }
        }
        right=k;
    }
}
```

Insertion Sort

Ý tưởng

Thuật toán Insertion Sort sắp xếp dựa trên tư tưởng là không gian cần sắp xếp đã được sắp xếp một đoạn và ta chỉ cần thêm giá trị mới vào không gian này sao cho không gian mới được sắp xếp.

Giả sử i phần tử đầu tiên $a_0, a_1, \ldots, a_i - 1$ đã có thứ tự.

Tìm cách chèn phần tử a_i vào vị trí thích hợp của đoạn đã được sắp để có đoạn mới $a_0, a_1, \ldots, a_i$ trở nên có thứ tự.

Các bước thực hiện:

Có n – 1 lần chèn

Ở mỗi lần chèn ta phải:

1. Tìm kiếm vị trí chèn hợp lệ (vị trí j).
2. Tuần tự dời các phần tử từ vị trí j trở đi xuống 1 vị trí về cuối mảng.
3. Đưa phần tử cần chèn vào vị trí j.

Độ phức tạp thuật toán

- **Best case** : Mỗi lần lặp sẽ có n – 1 bước so sánh. Số lần đổi chỗ là 0. Độ phức tạp sẽ là $O(n-1)$.
- **Worst case** :

Số lần so sánh tối đa trong lần chèn thứ i là $C(n)$.

Số lần chèn tối đa thứ i là $M(n)$

- **Average case** : $O(n^2)$.

Giải thuật mẫu

```
void insertionSort(int *a,int n)
{
    for(int i = 1; i < n; i++){

        int x = a[i];
        int j;
        for(j = i - 1; j >= 0  && a[j] > x; j--){

            //So sánh với phần tử trước đó (i-1), nếu bé hơn thì bắt đầu dời chỗ.
            //Dời chỗ cho đến khi gặp phần tử nhỏ hơn phần tử thứ i hồi nãy (x).
            a[j + 1] =a [j];
        }

        //Sau đó chèn phần tử i hồi nãy (x) vào vị trí đã tìm ở vòng lặp trên.
        a[j + 1] = x;
    }
}
```

Quick Sort

Ý tưởng

Gồm hai phần: Phân hoạch và sắp xếp, dựa trên ý tưởng chia để trị.

- **Bước 1** : Chọn tùy ý một phần tử a[k] trong dãy là phần tử nút trục (pivot) (l . Giả sử chọn ở giữa.

```
x = a[k];
i = L;
j = R;
k = (L + R) / 2;
```

Nếu $L \geq R$ (dãy có ít hơn 2 phần tử) kết thúc, dãy đã được sắp xếp.

- **Bước 2** : Ngược lại thì tạo vòng lặp phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ. (Nói cách khác là phân hoạch).
    - Trong khi (a[i] < x) i++;
    - Trong khi (a[j] > x) i--;
    - Nếu i <= j: Swap a[i],a[j];

Nếu i vẫn < j, lặp lại bước 2. Ngược lại i >= j thoát vòng lặp và gọi đệ qui.

- **Bước 3** : Gọi đệ qui đoạn bên trái pivot từ phần tử đầu hiện tại là L đến vị trí j;

Gọi đệ qui đoạn bên phải pivot từ phần tử i đến phần tử cuối hiện tại là R;

Bên trong các hàm đệ qui tiếp tục thực hiện tử bước 1 đến bước 3.

Bước đệ qui các mảng nhỏ hơn chính là bước trị, khi đi vào bước trị thì tiếp tục CẢ HAI bước chia để trị.

Có nhiều kỹ thuật phân hoạch, có thể tham khảo thêm slide của HCMUS.

Lưu ý : Việc chọn pivot là ngẫu nhiên, có thể chọn đầu, cuối hoặc ở giữa. Tuy nhiên chọn pivot ở đầu hoặc cuối trong một số trường hợp mảng gần như được sắp sẽ dẫn đến Worst case. Do đó chọn pivot ở giữa là chấp nhận được trong phần lớp các trường hợp. Mặc dù vậy, nếu chọn pivot ở cuối (hoặc đầu) thì việc phân hoạch có đôi chút sửa đổi. Có thể tham khảo ở đây.

Ngoài ra, không phải lúc nào cũng nên chọn phần tử có GIÁ TRỊ trung bình (average). Vì giá trị trung bình này không đại diện cho sự phân phối đồng đều của các phần tử trong mảng. Mà phần tử tốt nhất là phần tử có GIÁ TRỊ trung vị (median). Dẫu vậy, tìm trung vị của một dãy không hề đơn giản, nên chúng ta không tiếp cận theo hướng này.

Do đó, chúng ta lost-tolerant chọn ba phần tử đầu, cuối và giữa. Nếu dính vào trường hợp xấu nhất thì phải chấp nhận mặc dù xác suất xảy ra worst case là rất thấp. Thêm nữa, có thể sử dụng một trick là: so sánh ba phần tử đầu, cuối và giữa rồi lấy Median của ba vị trí này và chọn làm pivot.

Sự khác biệt giữa Quick và Merge : Merge mấu chốt ở bước tổng hợp, bước chia rất đơn giản. Tuy nhiên bước chia của Quick Sort rất phức tạp và quan trọng nhưng bước tổng hợp lại cực kì đơn giản.

Đầu vào – Đầu ra

- Input: Mảng A gồm n phần tử chưa sắp xếp, vị trí bên trái và vị trí bên phải của mảng hoặc phân hoạch.
- Output: Mảng A đã sắp xếp.

Độ phức tạp thuật toán

- **Best case** : $O(nlog_2(n))$.
- **Worst case** : $O(n^2)$.
- **Average case** : $O(nlog_2(n))$.

Giải thuật mẫu

```
void quickSort(int *a,int left, int right)
{
    //Chọn phần tử ở giữa làm phần tử pivot (nút trục)
    int pivot = a[(left + right) / 2];
    int i = left, j = right;

    //Phân hoạch mảng
    while(i < j)
    {
        while(a[i] < pivot) i++;
        while(a[j] > pivot) j--;
        if(i <= j)
            swap(a[i++],a[j--]);
    }

    //Gọi đệ qui sắp xếp cho các phân hoạch.
    if(j < left) quickSort(a,left,j);
    if(i > right) quickSort(a,i,right);
}
```

Merge Sort

Ý tưởng

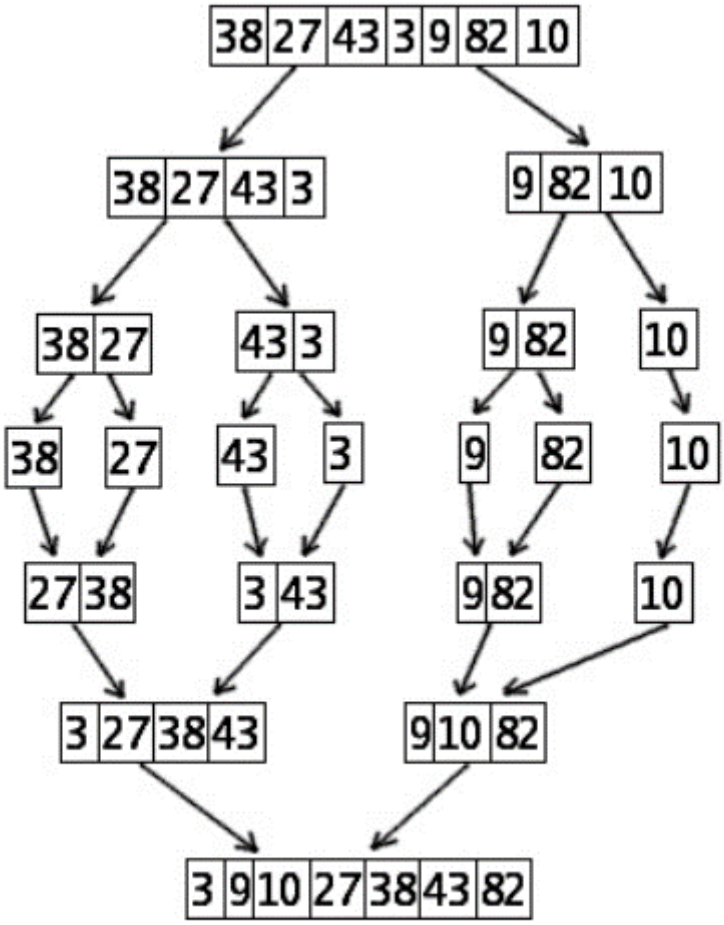
Thuật toán Merge Sort được chia thành hai phần.

Phần đầu tiên, chia các mảng thành hai không gian con, nếu các không gian con này có nhiều hơn một phần tử thì tiếp tục chia đôi. Ngược lại có duy nhất một phần tử hoặc không có phần tử nào (trong trường hợp dãy lẻ) thì bắt đầu trộn lại (gọi đệ qui).

Phần thứ hai chính là quá trình trộn, trong quá trình trộn sẽ kết hợp sắp xếp mảng.

Trộn 2 mảng con **đã được sắp xếp** thực hiện như sau:

1. Chọn phần tử min ở vị trí đầu của một trong hai mảng, xếp vào mảng cần trộn.
2. Phần tử nào đã xếp vào thì xóa đi, vị trí đầu của mảng là phần tử tiếp theo.
3. Nếu chưa đến cuối mảng thì lặp lại bước 1. Nếu đã đến cuối của một mảng (luôn xảy ra một mảng đã sắp hết và một mảng thì chưa), thì thêm toàn bộ mảng kia vào mảng cần sắp.



Nhận xét : Không tối ưu bộ nhớ vì dùng mảng tạm trong quá trình trộn. Nhanh hơn Quick Sort vì thời gian thực hiện Merge Sort có bậc nhỏ hơn $O(nlog_2(n))$, còn trong trường hợp tốt nhất Quick Sort mới có độ phức tạp là $O(nlog_2(n))$. Thường dùng Merge Sort để sắp khối dữ liệu lớn ở bộ nhớ ngoài.

Một phiên bản khác của Merge Sort không dùng đến việc chia mảng là Bottom - Up Merge Sort. Thuật toán này sẽ trộn các phần tử liền kề với nhau rồi mở rộng ra. Chẳng hạn như nó sẽ trộn 2 phần tử liên tiếp thành mảng con 2 phần tử. Sau đó nó sẽ trộn tiếp 2 mảng gồm 2 phần tử với nhau thành mảng 4 phần tử. Cứ như thế cho đến khi trộn hết mảng cũng là lúc mảng đã được sắp xếp.

Ngoài ra còn có Natural Merge Sort, thuật toán này sẽ không chia trực tiếp mà chẳng cần quan tâm đến thứ tự đã sắp xếp như Merge Sort. Ở đây nó sẽ xem xét các đường chạy (dãy đã có thứ tự). Trong khi Merge Sort cứng nhắc về số lần phân hoạch dựa vào chiều dài dãy là k, thì Natural Merge Sort sẽ dựa vào số đường chạy và mảng được sắp xếp là mảng chỉ có một đường chạy. Thực tế, người ta sử dụng Natural nhiều hơn trong trường hợp dãy đã sắp xếp một phần nào đó.

Một scenario điển hình là sắp xếp dữ liệu có kích thước lớn, người ta sẽ đưa nó vào bộ nhớ trong sắp xếp một phần rồi đem ra bộ nhớ ngoài để sử dụng Natural Merge Sort.

Do tính chất Non-inplace, thuật toán Merge Sort thường cần dùng bộ nhớ tạm trong quá trình thực thi, vì vậy mà thuật toán này sẽ sử dụng trong các cấu trúc dữ liệu khác mảng chẳng hạn như Linked List hoặc File.

Đầu vào – Đầu ra

- Input: Mảng A gồm n phần tử chưa sắp xếp, vị trí bên trái và vị trí bên phải của mảng hoặc phân hoạch.

- Output: Mảng A đã sắp xếp.

Độ phức tạp thuật toán

Độ phức tạp của phần merge là $O(n+m) = O(n)$ (n,m là kích thước hai mảng con cần trộn), và phần chia là $O(log_2 n)$

Best case, Worst case, Average case: $O(nlog_2(n))$.

Giải thuật mẫu

Phần chia mảng

```
void mergeSort(int *a,int left, int right)
{
    if(left <= right) return;
    //Tìm vị trí giữa mảng để chia

    int mid = (left + right) / 2;
    mergeSort(a,left,mid);
    mergeSort(a,mid + 1,right);
    merge(a,left,mid,right);
}
```

Phần trộn mảng

```
voidmerge(int *a, intleft, intmid, intright)
{
    int *temp = new int[right - left + 1];
    //Mảng thứ nhất từ left đến mid, mảng thứ hai từ mid + 1 đến right
    int k = 0;
    int i = left;
    int j = mid + 1;

    //?Khi hai mảng con chưa xét đến phần tử cuối
    while (i <= mid && j <=right)
    {
        //?So sánh phần tử đầu mỗi mảng và cho vào mảng phụ
        if (a[i] <= a[j])
        {
            temp[k++] =a[i++];
        }
        else
        {
            temp[k++] =a[j++];
        }
    }

    //?Chép các phần tử còn lại (nếu còn)
    while (i <= mid)
```

```
        {
            temp[k++] =a[i++];
        }
        while (j <= right)
        {
            temp[k++] =a[j++];
        }

        //?Chép mảng phụ lại vào mảng chính
        for (int i = left; i <= right; i++)
        {
            //Index của a tính từ left
            a[i] =temp[i - left];
        }
    }
```

Heap Sort

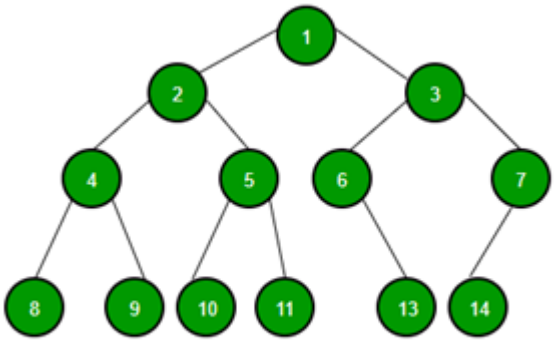
Ý tưởng

Thuật toán Heap Sort được chia làm hai phần.

Phần đầu tiên là xây dựng Max Heap từ mảng đầu vào.

Heap là gì? Heap là cây nhị phân hoàn chỉnh.

Cây nhị phân hoàn chỉnh là gì? Cây nhị phân là cây mỗi nốt đều có 0 hoặc 2 con.

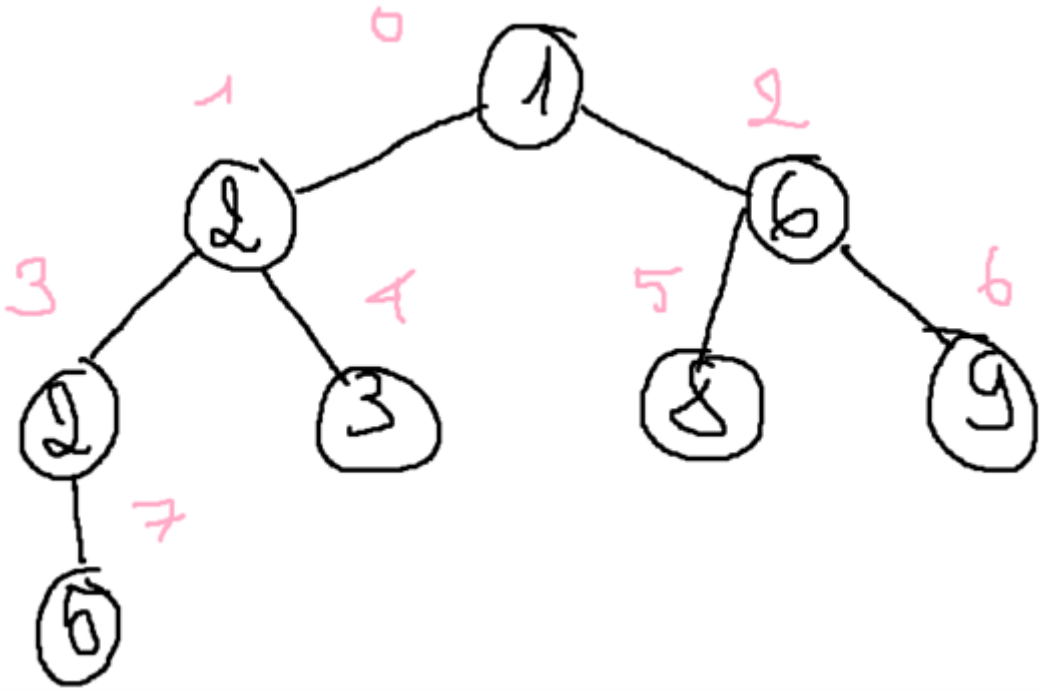


Hai phần tử con của một phần tử i bất kỳ là 2i+1 và 2i+2 (nếu phần tử đầu là 0) hoặc 2i và 2i+1 (nếu phần tử đầu là 1), là các phần tử **liên đới**.

Max Heap là Heap mà mỗi nốt đều lớn hơn các nốt con của nó. Phần này chính là đi xây dựng Max Heap từ phần tử giữa mảng. Theo hình trên thì các số chính là vị trí index trong mảng. Các số này được xây dựng trên việc duyệt cây theo các mức.

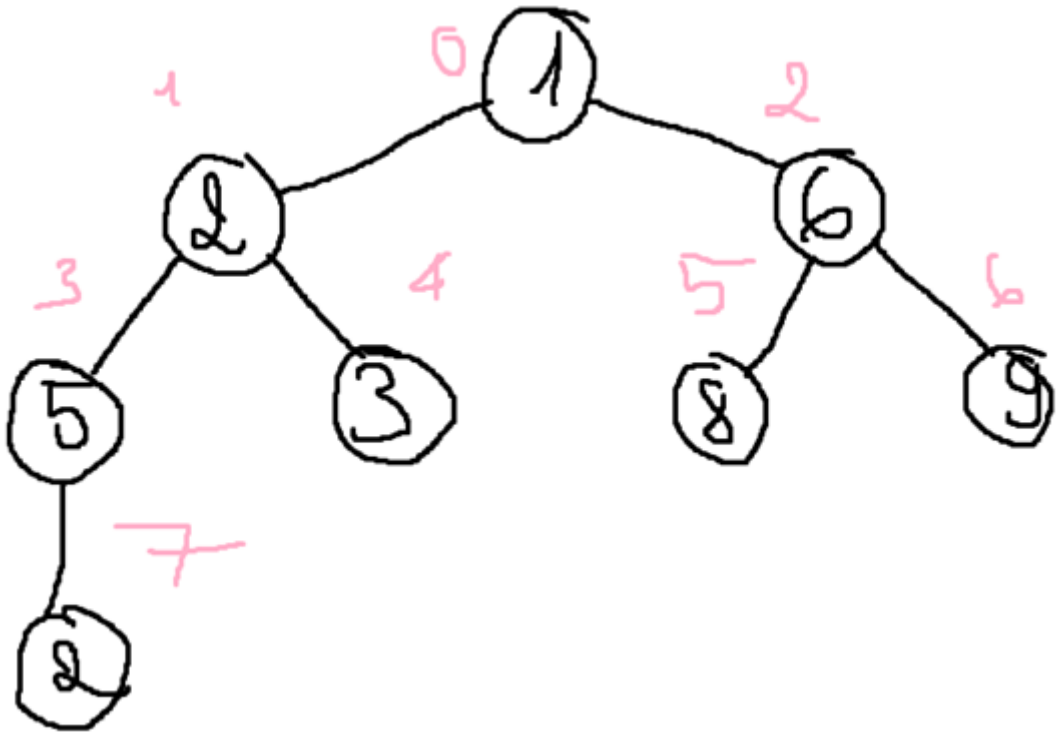
Ví dụ Max Heap 11 phần tử: 9, 8, 7, 5, 6, 3, 2, 4, 1, 1, 2. Có thể thấy với mỗi phần tử thứ i, nó sẽ lớn hơn các phần tử ở vị trí 2i+1 và 2i+2. Min Heap 7 phần tử 1, 2, 2, 3, 4, 5, 7.

Ví dụ có mảng [1, 2, 6, 2, 3, 8, 9, 5], thì cây đồ thị sẽ là:

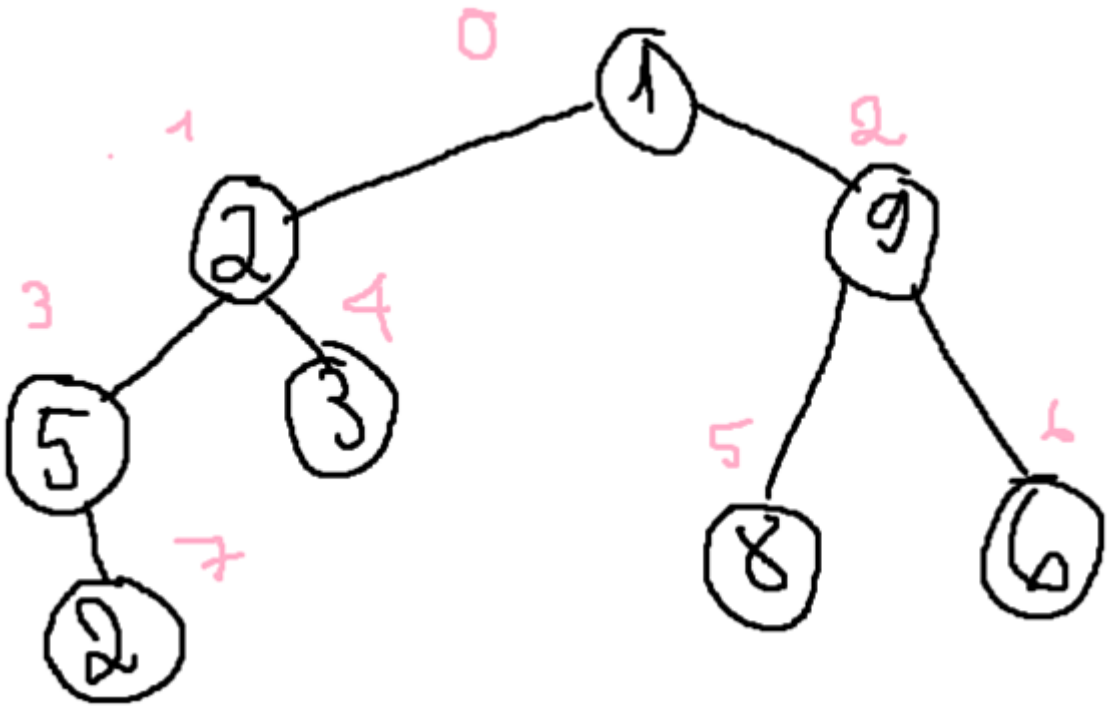


Ở ví dụ trên, đầu tiên sẽ vun đống phần tử giữa mảng, index sẽ là n/2 – 1 = 8/2 – 1 = 3. Mà a[3] là 2. Chúng ta sẽ hoán vị một phần tử với node con của nó nếu giá trị của node đó bé hơn node con. Và nếu cả hai node con đều lớn hơn, ta sẽ hoán vị với node con lớn nhất. Theo ví dụ trên thì ta có một hoán vị là (2-5).

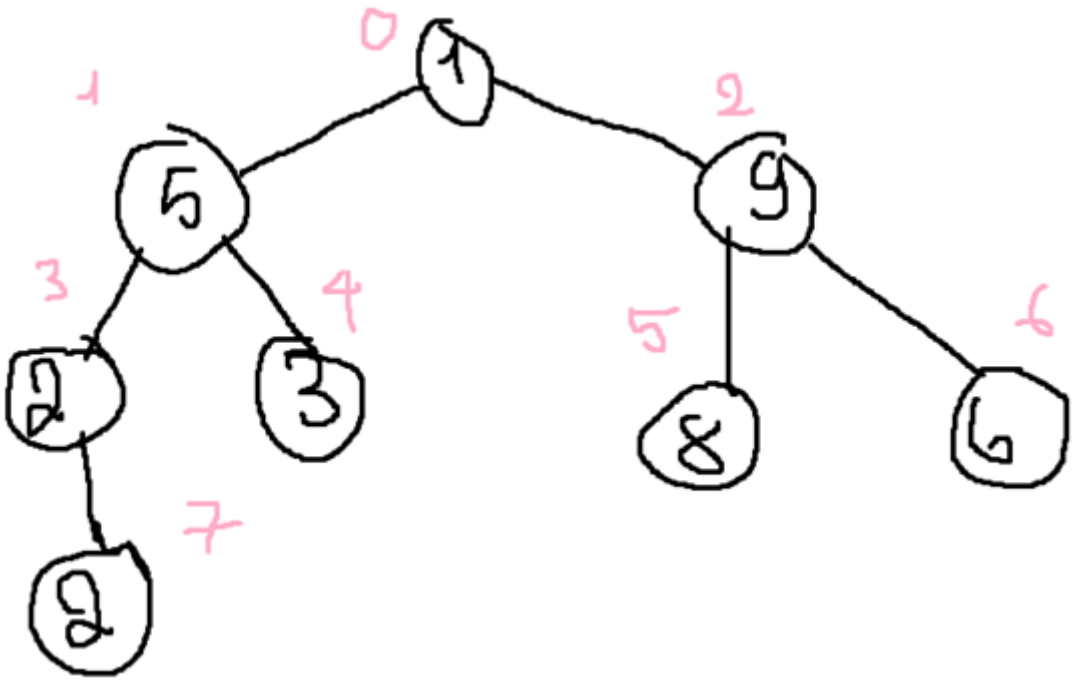
Các phần tử bị hoán vị phải được vun đống ở vị trí mà nó hoán vị đến. Việc này để vun đống lại Heap sau khi có sự ảnh hưởng gây ra bởi việc swap. ở trên ta sẽ xét vun đống tại a[7], nhưng không có gì xảy ra vì nốt này không có con.



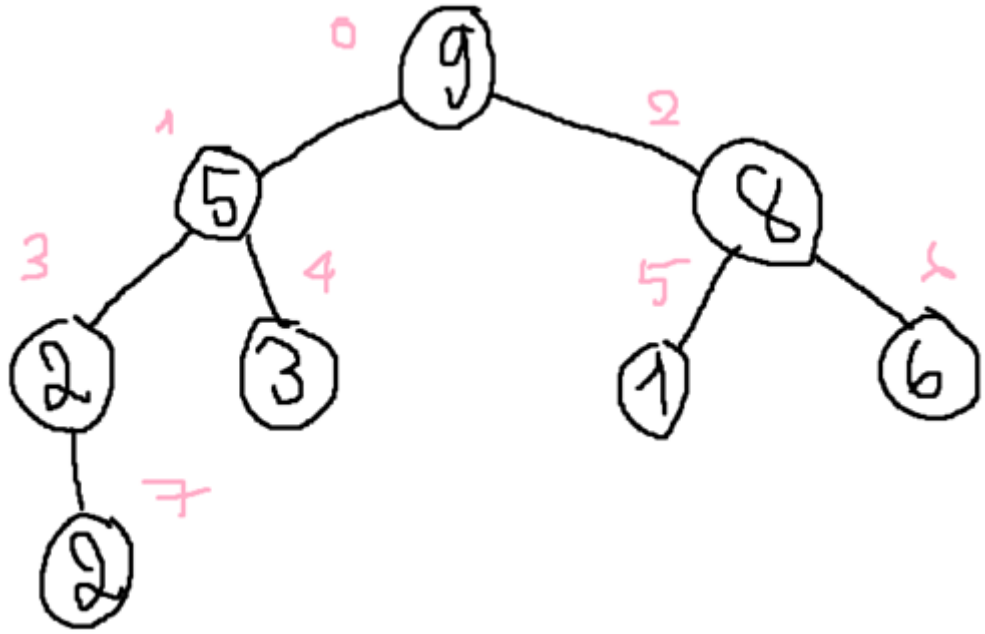
Tiếp theo sẽ xét vun đống tiến dần về đầu mảng, ta xét a[2] = 6. Sẽ có hoán vị (6-9). Xét đệ quy tại a[6] nhưng không có gì xảy ra do nốt không có con.



Xét a[1] = 2, ta sẽ hoán vị (2-5) và xét a[3], nhưng do 2 = 2 nên không vun đống tiếp.



Cuối cùng ta xét a[0] = 1, hoán vị (1-9, 1-8). Được kết quả là:



Mảng của chúng ta sau khi vun đống: [9, 5, 8, 2, 3, 1, 6, 2].

Phần thứ hai là sắp xếp. Ở phần này, khi đã xây dựng được Max Heap, chúng ta sẽ sắp xếp.

Ở phần sắp xếp này, chúng ta sẽ đổi chỗ phần tử đầu và phần tử cuối mảng. Sau đó sẽ xóa đi phần tử cuối mảng và tiếp tục Heapify (vun đống) như ở bước 1 (phần tử đầu hồi nãy có giá trị lớn nhất). Nhưng không cần Heapify toàn bộ, chỉ **xét phần tử đầu tiên** và build Heap ở đó do các phần tử còn lại đã được Heapify rồi.

Heap Sort là một thuật toán **cải tiến của Selection Sort**. Nhờ sử dụng cấu trúc Heap mà Heap Sort có số lần so sánh ít hơn Selection, nó chỉ tốn O(nlog(n)) chi phí so sánh.

Đầu vào – Đầu ra

- Input: Mảng A gồm n phần tử chưa sắp xếp.
- Output: Mảng A đã sắp xếp.

Độ phức tạp thuật toán

Best case: $O(nlog_2(n))$

Worst case: $O(nlog_2(n))$

Average case: $O(nlog_2(n))$

=> Consistency

Giải thuật mẫu

(Tham khảo GeeksforGeeks)

Phần Heapify (vun đống)

```
// To heapify a subtree rooted with node i, which is an index in arr.
// n is size of heap
```

```
void heapify(int* arr, int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

Phần sắp xếp

```
// main function to do heap sort
void heapSort(int* arr, int n)
{
    // Build heap (rearrange array) from middle to beginning of the array
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Advantage là một thuật toán hiệu quả, sử dụng ít bộ nhớ và ổn định. Ngoài ra thì cũng có thể tìm top k trong n như Selection Sort.

Heap Sort dựa vào việc so sánh để sắp xếp. Và sự so sánh này trên cấu trúc Heap không bị ảnh hưởng bởi kiểu dữ liệu (int, float, bool, char,...). Ngoài ra, đối với chuỗi ký tự, còn có **Dictionary-based.**

Ví dụ "been" > "ant", "been" < "boy", "Been" < "been". **Disadvantage** là một loại Unstable Sort, nếu dữ liệu quá lớn thì sẽ không hoạt động hiệu quả bằng Merge Sort.

Conclusion

Ta cần phân tích một chút lý do tại sao ba thuật toán Quick, Merge và Heap lại là $O(n.log_2n)$.

Nếu để ý, dễ dàng thấy được rằng cả ba thuật toán này đều có các bước chia đôi không gian sắp xếp ra để thao tác. Các bước để chia đôi một danh sách bất kỳ là log_2n. Lý do là vì hàm log_2n là hàm biểu thị cho một số nào đó, cần thiết để chia một danh sách n phần tử thành cơ số của nó (cơ số 2). Và cũng nói luôn, trong khoa học máy tính, chúng ta chỉ xét cơ số hai đối với hàm logarith. Ngược lại với nó là hàm mũ, nó biểu thị cho số lần để nhân tích lũy các cơ số lên để thành một số n nào đó. Nói cách khác, log_2n biểu thị cho số lần cần thiết để chia danh sách ban đầu thành nhiều phần nhỏ cho tiện việc xử lý.

Tiếp sau đó là việc xử lý. Trong hầu hết các công việc xử lý, như hàm Merge, thuật toán bắt buộc phải chạm tới từng phần tử trong bất kỳ lần trộn nào. Tức là công việc trộn này tốn O(n) thời gian để thực hiện. Quick sort cũng vậy, ở mỗi bước chúng ta đều phải duyệt các phần tử để đổi chỗ cho nhau. Tương tự đối với Heap sort.

Do vậy, với mỗi lần chia danh sách ban đầu ra làm hai, chúng ta phải duyệt qua n phần tử. Từ đó mà độ phức tạp tổng cộng sẽ là $O(n.log_2n)$.

Radix Sort

Ý tưởng

Radix – cơ số. Thuật toán này sắp xếp dựa trên cơ số (2, 8, 10, 16), mỗi cơ số có một cách biểu diễn riêng cho các giá trị phụ thuộc vào số lượng ký số (số ký tự để biểu diễn giá trị). Và mỗi bucket trong Radix Sort sẽ lưu các giá trị theo một loại ký số riêng biệt. Do đó nếu giá trị không thể hiện dưới dạng một số cơ số nhất định nào đó (chẳng hạn số thực) thì việc sắp xếp sẽ khá khó khăn.

Điểm mạnh của Radix Sort là nhanh, chỉ phụ thuộc vào số lượng ký số. Chẳng hạn có n phần tử nhưng phần tử có chiều dài ký số lớn nhất là 7, thì chỉ cần 7 lần sắp xếp.

Điểm yếu là số lượng bộ nhớ cần dùng rất lớn, nếu cơ số là 10, thì mảng sử dụng để lưu ký số là 10. Hơn nữa, nếu có n phần tử trong số các bucket, thì mảng cần sử dụng sẽ là 10*n.

Ứng dụng thường gặp của Radix Sort là sắp xếp thư từ dựa trên mã bưu chính (Postal Code hoặc ZIP Code)

Thay vì sort GIÁ TRỊ, chúng ta sẽ sort theo CHỮ SỐ của cơ số. Radix Sort có thể sử dụng Counting Sort làm hàm con của nó. Radix Sort là một thuật toán stable.

Chẳng hạn ta có một dãy số dưới đây:

$$129, 450, 356, 118, 928, 323, 875, 223$$

Đầu tiên, chúng ta dùng Counting Sort, sắp xếp các số dựa vào hàng đơn vị, số nào có hàng đơn vị lớn hơn thì đem ra sau, nhỏ hơn thì đem về trước.

Ta gạch dưới các chữ số hàng đơn vị.

$$129, 450, 356, 118, 928, 323, 875, 223$$

Sắp xếp dựa trên hàng đơn vị.

$$450, 323, 223, 875, 356, 118, 928, 129$$

Chúng ta có thể thấy thứ tự giữa các số có cùng hàng đơn vị vẫn không đổi. Sau đó ta gạch dưới các chữ số hàng chục và tiếp tục sắp xếp.

$$118, 323, 223, 928, 129, 450, 356, 875$$

Tiếp tục đối với hàng trăm.

$$118, 129, 223, 323, 356, 450, 875, 928$$

Kết quả thu được chính là mảng đã sắp xếp.

Đầu vào - Đầu ra

Tương tự các loại Sort không cần chia mảng.

Độ phức tạp thuật toán

Xét riêng Counting Sort, cần tốn n lần duyệt qua n phần tử. Đồng thời các key cần xem xét phụ thuộc vào hệ cơ số, nếu hệ cơ số là 10 thì k = 10 (các số chạy từ 0 đến 9), như ở ví dụ trên.

Tổng độ phức tạp thuật toán của mỗi lần thực hiện Counting Sort là O(n+k). Và chúng ta thực hiện Counting Sort phụ thuộc vào số chữ số tối đa của phần tử. Nếu phần tử có 3 chữ số như ví dụ trên thì Counting Sort thực hiện 3 lần, d chữ số thì thực hiện d lần.

Vậy độ phức tạp tổng quát là $O(d * (n + k))$.

Xét trong các trường hợp cụ thể khi giới hạn $k \leq nc$ và b là n. Thì độ phức tạp là $O(n)$.

Giải thuật mẫu

(Tham khảo GeeksforGeeks)

Phần Counting Sort

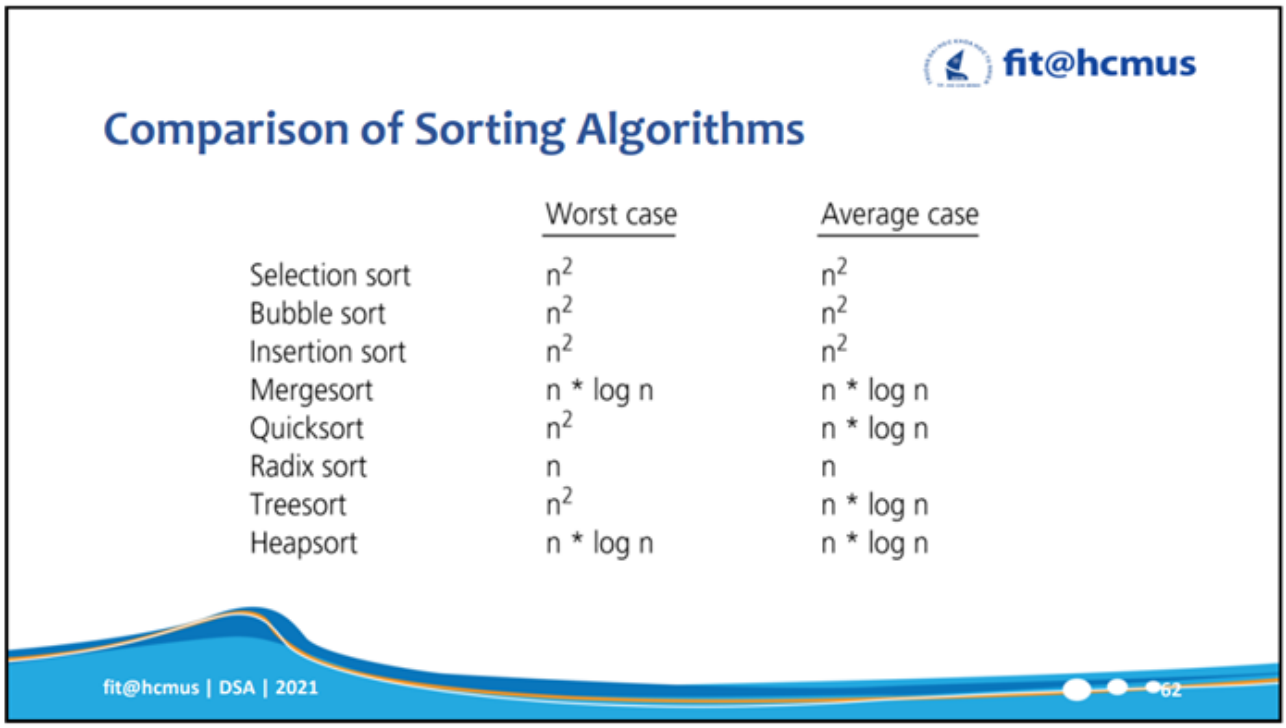
```
void countSort(int arr[], int n, int exp)
{
    int *output = new int[n]; // output array
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
```

```
    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Phần Radix Sort

```
// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

So sánh giữa các thuật toán


fit@hcmus
Comparison of Sorting Algorithms
Worst case
Average case
Selection sort
n2
n2
Bubble sort
n2
n2
Insertion sort
n2
n2
Mergesort
n * log n
n * log n
Quicksort
n2
n * log n
Radix sort
n
n
Treesort
n2
n * log n
Heapsort
n * log n
n * log n
fit@hcmus | DSA | 2021
62