- Có hai loại giải thuật sắp xếp được dùng phổ biến trong khoa học máy tính là sắp xếp nội (internal sorting) và sắp xếp ngoại (external sorting):

* Với loại sắp xếp nội thì toàn bộ dữ liệu cần sắp xếp được đưa vào bộ nhớ trong, do vậy kich thước dữ liệu cần sắp xếp không lớn, tuy nhiên thời gian sắp xếp được thực hiện rất nhanh.
* Với loại sắp xếp ngoại thì chỉ một phần nhỏ dữ liệu cần sắp xếp được đưa vào bộ nhớ trong, phần lớn dữ liệu còn lại được lưu ở bộ nhớ ngoài (như đĩa từ băng từ, ...), kích thước dữ liệu cần sắp xếp lúc này rất lớn (phụ thuộc vào kích thước của đĩa từ, băng từ, ...), và thời gian sắp xếp thực hiện rất chậm.

- Internal memory: nói về RAM

- External memory: nói về đĩa cứng, HDD, SSD,…

- Sự khác biệt giữa Internal memory và External memory:

+ Internal memory:

* Tốc độ truy xuất nhanh hơn

+ External memory:

* Tốc độ truy xuất chậm hơn

- In-place (algorithm)

- Stable (algorithm)

5 THUẬT TOÁN SẮP XẾP THÔNG DỤNG

1. Selection Sort (Sắp xếp chọn)

a. Ý tưởng thuật toán

- Duyệt từ đầu đến phần tử kế cuối danh sách, duyệt tìm phần tử nhỏ nhất từ vị trí kế phần tử đang duyệt cho đến hết danh sách, sau đó đổi vị trí của phần tử nhỏ nhất đó với phần tử đang duyệt và cứ tiếp tục cho đến hết danh sách.

b. Các bước giải quyết

- Cho mảng A có n phần tử chưa được sắp xếp. Cụ thể các bước của giải thuật này áp dụng trên mảng A như sau:

B1: Gán i = 0

B2: Gán min = A[i] và j = i + 1

B3: Nếu j < n:

* Nếu A[j] < min thì min = A[j]
* j = j + 1
* Quay lại B3

B4: Đổi chỗ A[i] và min

B5: Nếu i < n-1:

* Đúng thì i = i + 1 và quay lại B2
* Sai thì dừng lại

c. Thuật toán hoàn chỉnh

void SelectionSort(int a[], int n)

{

int min = 0;

for (int i = 0; i < n - 1; i++)

{

min = i;// xem a[min]=a[i] là phần tử nhỏ nhất

for (int j = i + 1; j < n; j++)

{

if (a[j] < a[min])// Nếu a[j]<a[min] thì a[j] là nhỏ nhất

{

min = j;// Lưu lại vị trí a[min] vừa tìm được

}

}

if (a[min] != a[i])// Nếu a[min] không phải là a[i] ban đầu thì đổi chỗ hai phần tử

{

SWAP(a[i], a[min]);

}

}

}

d. Độ phức tạp thuật toán

- Đối với thuật toán sắp xếp chọn, do sử dụng 2 vòng lặp lồng vào nhau, độ phức tạp thời gian trung bình, của thuật toán này đều là O(n2). Thuật toán sắp xếp chọn mình cài đặt là thuật toán sắp xếp không ổn định, nó còn có một phiên bản khác cải tiến là thuật toán sắp xếp chọn ổn định.

- Trường hợp tốt nhất: O(n^2)

- Trường hợp xấu nhất: O(n^2)

e. Ưu điểm và nhược điểm

- Ưu điểm: Thuật toán chạy nhanh hơn khi mảng sắp xếp một phần

- Nhược điểm: Hiệu suất không cao

2. Heap Sort (Thuật toán sắp xếp vun đống)

a. Ý tưởng thuật toán

- Chúng ta đưa mảng A về dạng cấu trúc dữ liệu Binary Heap. Việc tìm kiếm tương tự như Selection Sort ta sẽ tìm phần tử lớn nhất và đưa vào cuối, tương tự cho các phần tử khác.

- Định nghĩa về Binary Heap:

* Đầu tiên chúng ta hãy xác định một Cây nhị phân hoàn chỉnh. Cây nhị phân hoàn chỉnh là cây nhị phân trong đó mọi cấp, ngoại trừ cấp cuối cùng, được lấp đầy hoàn toàn và tất cả các nút ở bên trái càng xa càng tốt
* Một Binary Heap là một cây nhị phân hoàn chỉnh trong đó các mục được lưu trữ theo một thứ tự đặc biệt sao cho giá trị trong nút cha lớn hơn (hoặc nhỏ hơn) so với giá trị trong hai nút con của nó. Cái trước được gọi là max heap và cái sau được gọi là min-heap. Heap có thể được biểu diễn bằng một cây hoặc mảng nhị phân.

- Vì một Binary Heap là một cây nhị phân hoàn chỉnh, nó có thể dễ dàng được biểu diễn dưới dạng một mảng và cách biểu diễn dựa trên mảng là hiệu quả về không gian. Nếu nút cha được lưu trữ ở chỉ mục I, nút con bên trái có thể được tính bằng 2 \* I + 1 và nút con bên phải bằng 2 \* I + 2 (giả sử việc lập chỉ mục bắt đầu từ 0).

b. Các bước giải quyết

- Cho mảng A có n phần tử chưa được sắp xếp, hãy sắp xếp mảng A tăng dần. Cụ thể các bước của giải thuật này áp dụng trên mảng A như sau:

* B1: chúng ta sẽ tạo ra một max heap từ dữ liệu đầu vào.
* B2: lúc này phần tử lớn nhất được lưu ở gốc của heap, swap(gốc, mục cuối cùng của heap) -> Ta có được một gốc của heap là phần tử lớn nhất của mảng.
* B3: lặp lại B2 khi kích thước của heap lớn hơn 1.

- Cách để xây dựng một heap:

VD: cho một mảng như sau a: 4, 10, 3, 5, 1

* Gốc của heap sẽ là 4 (tại i = 0)
* 2 child của 4 sẽ là 10 (left node = 2\*i + 1) và 3 (right node = 2\*i + 2)
* Cứ tiếp tục như thế ta sẽ có 10 là parent node của 5 và 1

4

(0)

3

10

(1) (2)

5

1

(3) (4)

Mảng a: 4 10 3 5 1

- Sau khi xây dựng xong một heap ta sẽ xây dựng max heap cho nó và làm theo các bước 1 ,2 ,và 3.

- Để xây dựng max heap ta làm như sau:

* Tìm last parent node bằng cách lấy (n/2-1) làm tròn xuống sẽ được vị trí last parent node gọi là lpn, trong trường hợp này lpn = a[1] = 10.
* Nếu child của lpn lớn hơn nó thì swap chúng với nhau.

Do 10 > 5 và 1 nên không có swap

* Tiếp đến là đi đến những parent tiếp theo

Lpn đang ở vị trí là i=1 thì parent của nó là i-1=0

Do left lớn hơn parent của nó nên ta sẽ swap

10

(0)

3

4

(1) (2)

5

1

(3) (4)

Ta xét tiếp tại phần tử vừa mới được swap là vị trí i=1 thấy 4 nhỏ hơn hơn child của nó là 5 nên ta sẽ tiếp tục swap

10

(0)

3

5

(1) (2)

4

1

(3) (4)

Mảng a: 10 5 3 4 1

* swap(gốc, cuối heap) là 10 và 1, sau đó bỏ phần tử cuối ra khỏi heap

1

(0)

3

5

(1) (2)

4

10

(3) (4)

Mảng a: 1 5 3 4 10

* Tiếp tục lấy gốc của heap (tại vị trí 0) đem đi so sánh với child của nó và tiếp tục tiếp tục như vậy cho đến cuối heap sau đó truy ngược trở lại (tương tự đệ quy)

- Ta cứ làm tuần tự những bước trên cho đến khi kích thước của heap bằng 1 là giải quyết xong bài toán.

c. Thuật toán hoàn chỉnh

> Hàm thực hiện vun đống trong Heap

void Heapify(int a[], int n, int i)

{

int LargestNode = i;// Khởi tạo một cái parent

int LeftNode = i \* 2 + 1;// Left của parent

int RightNode = i \* 2 + 2;// Right của parent

// So sánh parent với Left

if (LeftNode<n && a[LeftNode]>a[LargestNode])

{

LargestNode = LeftNode;

}

// So sánh parent với Right

if (RightNode<n && a[RightNode]>a[LargestNode])

{

LargestNode = RightNode;

}

// Cập nhật node lớn nhất nếu node lớn nhất không là parent ban đầu

if (LargestNode != i)

{

// Tiếp tục truy vết các node lớn nhất cho đến cuối Heap(đệ quy)

SWAP(a[LargestNode], a[i]);

// Recursion

Heapify(a, n, LargestNode);

}

}

> Hàm thực hiện HeapSort

void HeapSort(int a[], int n)

{

// Xây dựng Max Heap

for (int i = n / 2 - 1; i >= 0; i--)

{

Heapify(a, n, i);

}

// Trích xuất từng phần tử lớn nhất của Max Heap ứng với các gốc tương tứng

for (int i = n - 1; i > 0; i--)

{

// Đưa gốc hiện tại đến mục cuối cùng của Heap

SWAP(a[0], a[i]);

// Loại bỏ phần tử cuối ra khỏi Heap và bắt đầu truy vết từ gốc (vị trí i = 0)

Heapify(a, i, 0);

}

}

d. Độ phức tạp thuật toán

- Đối với thuật toán sắp xếp vun đống(Heap Sort), độ phức tạp về thời gian của heap là O(log(n)), độ phức tạp thời gian của tạo Heap và xây dựng Heap là O(n) vậy độ phức tạp thời gian tổng thể của Heap Sort là O(n\*log(n)).

- Heap sort là một thuật toán tại chỗ.

- Tùy loại mà triển khai hình thức sắp xếp này vì nó không ổn định, nhưng nó cũng có thể ổn định

- Trường hợp tốt nhất: O(nlog(n))

- Trường hợp xấu nhất: O(nlog(n))

e. Ưu điểm và nhược điểm

- Ưu điểm: Hiệu suất của thuật toán cao

- Nhược điểm: Code của thuật toán phức tạp, khó hiểu.

3. Merge Sort (Sắp xếp trộn)

Có 2 loại Merge Sort là: Internal Sorting và External Sorting

a. Ý tưởng thuật toán

- Sắp xếp trộn (merge sort) là một thuật toán dựa trên kĩ năng chia để trị, ý tưởng của thuật toán này là chia đôi mảng thành hai mảng con, sau đó sắp xếp hai mảng con và trộn lại theo đúng thứ tự, mảng con được sắp xếp theo cách tương tự.

b. Các bước giải quyết

- Giả sử left là vị trí đầu, right là vị trí cuối mảng đang xét, các bước cụ thể của thuật toán như sau:

Nếu left<right tức là mảng còn chia đôi được

B1: Tìm vị trí mid của mảng

B2: Sắp xếp mảng thứ nhất(từ left đến mid)

B3: Sắp xếp mảng thứ hai(từ mid+1 đến right)

B4: Trộn hai mảng đã sắp xếp với nhau

c. Thuật toán hoàn chỉnh

// Hàm trộn hai mảng

void Merge(int a[], int l, int mid, int r)

{

// Số phần tử mảng thứ nhất

int n1 = mid - l + 1;

// Số phần tử mảng thứ hai

int n2 = r - mid;

// Tạo hai mảng tạm để lưu hai mảng con

int\* a1 = new int[n1];

int\* a2 = new int[n2];

// Sao chép hai mảng con vào hai mảng tạm

for (int i = 0; i < n1; i++)

{

a1[i] = a[l + i];

}

for (int i = 0; i < n2; i++)

{

a2[i] = a[mid + 1 + i];

}

int i = 0, j = 0;

// k là vị trí hiện tại trong mảng a

int k = l;

// Trộn hai mảng theo đúng thứ tự

while (i < n1 && j < n2)

{

if (a1[i] <= a2[j])

{

a[k++] = a1[i++];

}

else a[k++] = a2[j++];

}

// Nếu mảng thứ nhất còn phần tử thì copy vào mảng a

while (i < n1)

{

a[k++] = a1[i++];

}

// Nếu mảng thứ hai còn phần tử thì copy vào mảng a

while (j < n2)

{

a[k++] = a2[j++];

}

// Xóa bộ nhớ hai mảng tạm

delete[]a1, a2;

}

// Hàm chia đôi mảng và gọi hàm trộn

void \_MergeSort(int a[], int l, int r)

{

// Kiểm tra xe còn chia đôi mảng được hay không

if (l < r)

{

// Tìm vị trí chính giữa của mảng

int mid = l + (r - l) / 2;

// Sắp xếp mảng thứ nhất từ vị trí l đến mid

\_MergeSort(a, l, mid);

// Sắp xếp mảng thứ hai từ vị trí mid + 1 đến r

\_MergeSort(a, mid + 1, r);

// Trộn hai mảng đã sắp xếp

Merge(a, l, mid, r);

}

}

// Hàm thực hiện sắp xếp trộn

void MergeSort(int a[], int n)

{

\_MergeSort(a, 0, n - 1);

}

d. Độ phức tạp thuật toán

- Về độ phức tạp, thuật toán Merge Sort có độ phức tạp thời gian trung bình là O(nlog(n)), về không gian, do sử dụng mảng phụ để lưu trữ, và 2 mảng phụ dài nhất là hai mảng phụ ở lần chia đầu tiên có tổng số phần tử bằng đúng số phần tử của mảng nên độ phức tạp sẽ là O(n). Sắp xếp trộn là thuật toán sắp xếp ổn định.

- Trường hợp tốt nhất: O(nlog(n))

- Trường hợp xấu nhất: O(nlog(n))

e. Ưu điểm và nhược điểm

- Ưu điểm: Hiệu suất của merge sort rất cao

- Nhược điểm: Code thuật toán này khá phức tạp

4. Quick Sort (Sắp xếp nhanh)

a. Ý tưởng thuật toán

- Sắp xếp nhanh (quick sort) hay sắp xếp phân đoạn (Partition) là là thuật toán sắp xếp dựa trên kỹ thuật chia để trị, cụ thể ý tưởng là: chọn một điểm làm chốt (gọi là pivot), sắp xếp mọi phần tử bên trái chốt đều nhỏ hơn chốt và mọi phần tử bên phải đều lớn hơn chốt, sau khi xong ta được 2 dãy con bên trái và bên phải, áp dụng tương tự cách sắp xếp này cho 2 dãy con vừa tìm được cho đến khi dãy con chỉ còn 1 phần tử.

b. Các bước giải quyết

B1: Chọn một phần tử làm chốt

B2: Sắp xếp phần tử bên trái nhỏ hơn chốt

B3: Sắp xếp phần tử bên phải nhỏ hơn chốt

B4: Sắp xếp hai mảng con bên trái và bên phải pivot

c. Thuật toán hoàn chỉnh

// Hàm sắp xếp phân đoạn

void \_QuickSort(int a[], int l, int r)

{

// Nếu mảng có một phần tử thì không cần sắp xếp

if (l >= r) return;

// Tìm phần tử làm chốt (phần tử pivot)

int mid = (l + r) / 2;

int pivot = a[mid];

// i là vị trí đầu đoạn, j là vị trí cuối đoạn

int i = l, j = r;

while (i < j)

{

// nếu phần tử bên trái nhỏ hơn pivot thì bỏ qua phần tử đó

while (a[i] < pivot)

{

i++;

}

// nếu phần tử bên phái lớn hơn pivot thì bỏ qua phần tử đó

while (a[j] > pivot)

{

j--;

}

// Sau khi kết thúc hai vòng while trên thì ta sẽ có

// a[i] lớn hơn pivot và a[j] nhỏ hơn pivot

if (i <= j)

{

// Nếu i khác j thì mới cần hoán đổi

if (i < j) swap(a[i], a[j]);

// Nếu i = j thì không cần hoán đổi

i++;

j--;

}

}

// Gọi đệ quy sắp xếp dãy bên trái pivot

\_QuickSort(a, l, j);

// Gọi đệ quy sắp xếp dãy bên phải pivot

\_QuickSort(a, i, r);

}

// Hàm sắp xếp chính

void QuickSort(int a[], int n)

{

\_QuickSort(a, 0, n - 1);

}

d. Độ phức tạp thuật toán

- Thuật toán sắp xếp nhanh không phải là thuật toán sắp xếp ổn định, tuy nhiên vẫn có thể cải tiến nó thành thuật toán sắp xếp ổn định. Độ phức tạp thời gian trung bình của thuật toán này là O(n\*log(n)).

- Trường hợp tốt nhất: O(n\*log(n))

- Trường hợp xấu nhất: O(n^2)

e. Ưu điểm và nhược điểm

- Ưu điểm: Tuỳ cách chọn pivot mà tốc độ của thuật toán nhanh hay chậm

- Nhược điểm: Code khá phức tạp

5. Radix Sort (Thuật toán sắp xếp theo cơ số)