Nguyễn Minh Đức 1712358

Lý thuyết Cấu trúc dữ liệu và giải thuật 17CTT3

Tuần 4

Phần trắc nghiệm

1) A

2) B

3) D

4) D

5) A

6) C

7) A

8) C

9) D

10) E

Phần tự luận

1)

Cây nhị phân dùng cho Heap Sort là cây nhị phân đầy đủ có chiều cao tối đa là log(n).

Để tính được độ phức tạp của thuật toán thì ta tính độ phức tạp của hàm Heapify trước, do hàm này được dùng xuyên suốt quá trình sắp xếp.

Hàm Heapify dùng cho việc sửa 1 vị trí bị sai trong cây => Heapify sẽ cố độ phức tạp trong trường hợp xấu nhất là O(h) (với h là chiều cao của cây) và bằng O(log(n)).

Heap Sort gồm 2 bước:

Bước 1: xây dựng Heap

Quá trình thực chất là sử dụng hàm Heapify lần lượt từ phần tử n/2-1 về đến phần tử đầu tiên của mảng => quá trình này có độ phức tạp:

O(log(2)+ log(3)+ … + log(n/2))

với log(2) + …+ log(n/2) gần bằng (n/4)*log(n/4)

=> quá trình trên sẽ tốn 1 thời gian O((n/4)*log(n/4)) = O(nlog(n))

Bước 2: Sắp xếp

Bắt đầu bằng việc hoán ví phần tử đầu tiên với phần tử cuối cùng của Heap (O(1)), giảm số phần tử của Heap đi 1 sau đó Heapify phần tử đầu tiên, lặp lại quá trình trên

đến khi mảng được sắp xếp hoàn toàn (lặp lại đến khi mảng chỉ còn 2 phần tử, hoán vị 2 phần tử, sắp xếp kết thúc).

Thời gian thực hiện hoán vị O(n-1) = O(n)

Tổng thời gian thực hàm Heapify

O(log(2) + log(3) +…+ log(n))

với log(2)+…+log(n) gần bằng (n/2)*log(n/2)

=> Tổng thời gian thực hiện trong bước 2 là O(nlog(n)+O(n)) = O(nlog(n))

Tổng kết lại 2 quá trình => Heap sort có độ phức tạp O(nlog(n))

2)

Ta chỉ cần sửa lại 1 chút chỗ so sánh a[i], a[j] với phần tử key

```
void quickSort(int *a, int l, int r)
{
   int key = a[(l+r)/2];
   int i = l, j = r;

   while(i <= j)
   {
      while(a[i] > key) i++;
      while(a[j] < key) j--;
      if(i <= j)
      {
         if (i < j)
            swap(int, a[i], a[j]);
         i++;
         j--;
      }
   }
   if (l < j) quickSort(a, l, j);
   if (i < r) quickSort(a, i, r);
}
```

3)

Nếu ta sắp xếp theo kiểu

< pivot | >= pivot (1)

(các phần tử nhỏ hơn pivot nằm bên trái vách ngăn, lớn hơn hoặc bằng nằm bên phải)

hoặc theo kiểu

<= pivot | > pivot (2)

thì theo kiểu (1) phân hoạch quick sort sẽ chia mảng ra làm 2 phần với số phần tử tương ứng là 1 | n-1,theo kiểu 2 thì sẽ được chia thành n-1 | 1. Bởi các phần tử đều bằng nhau nên ở 1 đầu tìm kiếm phần tử < pivot hoặc > pivot thì biến chạy sẽ chạy đến khi gặp đầu bên kia.

việc phân chia này cũng đúng cho các mảng con, nên sẽ có n-1 lần chia, với mỗi lần chia đều xảy ra n lần so sánh => độ phức tạp của Quick sort trong 2 kiểu phân chia này là O(n^2).

Nhưng nếu ta phân chia theo kiểu

<= pivot | >= pivot

thì sau 1 lần phân hoạch quick sort sẽ chia mảng ra làm 2 phần đều nhau n/2 | n/2 (do khi đi tìm phần tử <= hoặc >= pivot biến chạy ở 2 đầu sẽ di chuyển cùng lúc mỗi lần 1 đơn vị rồi ngay lập tức hoán vị cho nhau, dù việc hoán vị này là rất lãng phí)

nên sẽ chỉ có log(n) lần chia, với mỗi lần phân chia thì có n phép so sánh

=> quick sort trong kiểu phân chia này sẽ có độ phức tạp O(nlog(n)). Mặc dù hệ số đứng trước nlog(n) sẽ lớn hơn nhiều so với 2 kiểu chia phân hoạch ban đầu.

4)

Đoạn phân hoạch tồn tại rất nhiều lỗi sai.

Ví dụ cho mảng 2 phần tử a[] = {2, 1}

=> x = a[2/2] = a[1] = 1;

vòng lặp đầu tiên i sẽ dừng lại tại i = 0 (do 2 > x), j dừng lại tại j = 1,

sau hoán vị thì mảng a[] = {1, 2}, nhưng thuật toán chưa dừng do không tăng i và giảm j sau khi hoán vị nên i < j, vòng lặp sẽ chạy 1 lần nữa. Khi này i = 1, và j = 0 và xảy ra hoán vị không mong muốn, mảng a[] = {2, 1} như ban đầu. Khi này vòng lặp kết thúc bởi i đã lớn hơn j.

5)

6)

Trước tiên ta xét 1 ví dụ về sắp xếp tăng dần mảng a 3 phần tử.

Mục đích của việc so sánh là tìm ra thứ tự giữa 2 phần tử với nhau.

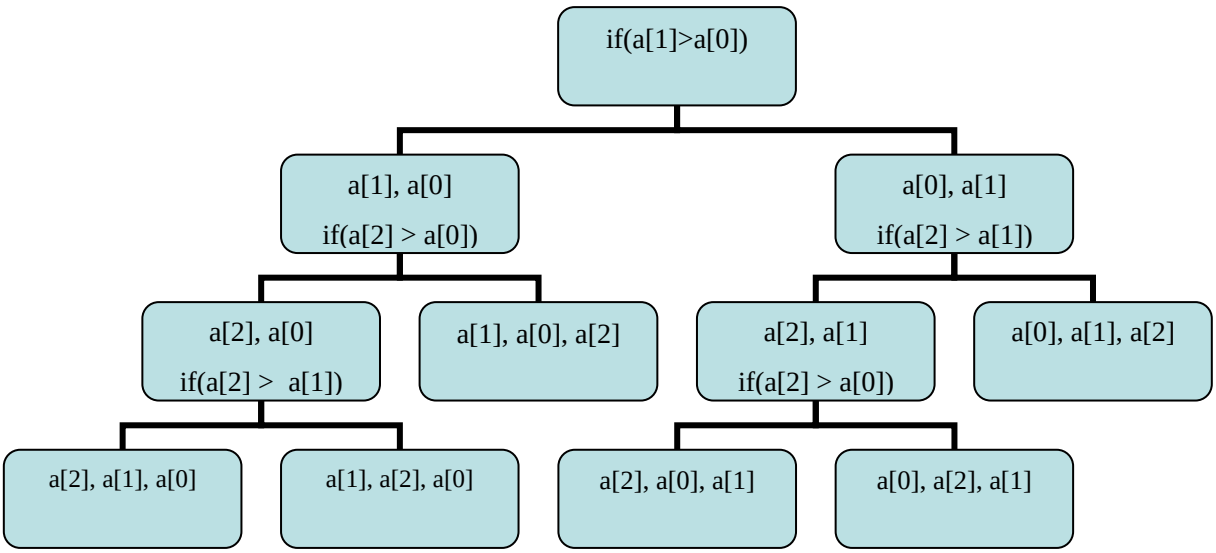
Ví dụ mới đầu ta so sánh a[1] và a[0], nếu a[1] > a[0] trả về true, ta nói a[1] đứng sau a[0] và có thứ tự a[0], a[1].

Ta cần phải tìm vị trí cho a[2], sẽ chỉ cần 1 phép sánh nếu a[2] > a[1] trả về true, thứ tự đúng là a[0], a[1], a[2], nhưng nếu a[2] > a[1] trả về false thì ta biết được a[2] đứng

trước a[1] nhưng không biết a[2] đứng trước hay sau a[0], ta phải so sánh a[2] > a[0] để tìm ra đáp án.

Để mô tả quá trình dùng các phép so sánh tìm ra thứ tự đúng cho tất cả phần tử, ta có thể dùng decision tree (cây quyết định)

(Từ gốc đi xuống nếu phép so sánh trả về true, đi về bên phải, ngược lại, đi về bên trái)



Node leaf chính là đáp án.

Đoạn đường đi từ node root đến leaf chính là thời gian của thuật toán, ở decision tree trên thấy có vẻ như có những đường đi ngắn hơn những đường còn lại, giống như cách chúng ta chọn ra 2 phần tử để so sánh với nhau, nếu ta chọn 2 phần tử liên tiếp so sánh với nhau, và đầu vào là 1 mảng đã được sắp xếp sẵn thì sẽ chỉ mất có n-1 lần so sánh, nhưng nếu mảng đầu vào là 1 mảng ngẫu nhiên thì có khả năng cao ta phải tốn nhiều thời gian hơn.

Nhưng dù chiến thuật của ta là gì đi chăng nữa, thời gian sắp xếp cũng là chiều cao của decision tree, vì decision tree phải chứa tất cả các khả năng có thể xảy ra của 1 mảng đã sắp xếp ở các node leaf của mình, và 1 nội dung quan trọng nữa là decision tree là 1 cây nhị phân (do các nhánh đều chia ra true hoặc false, chỉ có 2 nhánh cho mỗi node) nên ta có thể tính được chiều cao tối đa của decision tree nhờ vào việc tính số khả năng có thể xảy ra (tức tính số lượng node leaf có thể có).

Nếu decision tree có m node leaf ta có thể kết luận ngay chiều cao của cây này tối đa là log(m).

Ta lấy m node leaf chia 2 sẽ được số lượng node ở level thấp hơn, chia 2 1 số lần nhất định sẽ được kết quả 0, số lần chia 2 chính là chiều cao của cây, vì vậy 2^h = m nên => h = log(m). Rất ngạc nhiên ở đây là nếu tổng số node của cây là n thì chiều cao của

cây cũng là log(n) bằng với việc lấy log của số lượng node leaf. Đúng vậy, đặt số lượng node leaf là n^h, ta có thể chứng minh 2^h = 1+ 2^0+2^1+...+2^(h-1), tức số lượng node leaf có thể bằng tất cả lượng node còn lại cộng thêm 1, nên việc lấy log sẽ không tăng thêm 1 đơn vị nào.

Với 1 mảng n phần tử, ta sẽ có n! các sắp xếp các phần tử và cũng là tất cả các khả năng có thể xảy ra ở node leaf của decision tree. => chiều cao của cây là log(n!)

Ta có thể xấp xỉ kết quả trên bằng 1 cách mà trong khoa học máy tính hay sử dụng

log(n!) = log(1) + log(2) +...+log(n) gần bằng (n/2)*log(n/2)

và bằng (n/2)*log(n) – n/2 = O(nlog(n))

=>Dù với chiến thuật nào đi chăng nữa, các thuật toán sử dụng các phép so sánh trong việc sắp xếp dữ liệu thì sẽ không thể có thời gian chạy tốt hơn O(nlog(n)).