Algorithm analysis - Merge sort + Heap sort

Team 08 - DTA

March 27^{th} 2021

1 Phân tích độ phức tạp thuật toán Merge Sort

Thuật toán Merge Sort có thể được mô tả bằng lời như sau:259259259

- *Bước 1*: Nếu chỉ có một phần tử trong mảng thì kết thúc việc gọi đệ quy.
- *Bước 2*: Sử dụng đệ quy chia mảng ra thành hai nửa thành phần độc lập.
- *Bước 3*: Gộp các mảng nhỏ thành những mảng lớn hơn theo thứ tự.

Qua đó, ta có thể xây dựng được hệ thức truy hồi cho phương trình đệ quy của thuật toán như sau:

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(\frac{n}{2}) + d(n) \text{ với } n \geq 2 \end{cases}$$

Xét phương trình $T(n)$, ta có:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + d(n) \\ &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2n \\ &\dots \\ &= 2^k T\left(\frac{n}{2^k}\right) + kn \end{aligned} \tag{1}$$

Chứng minh (1) bằng quy nạp, ta có:
Với $k = 1$:

$$(1) \Leftrightarrow T(1) = 2T\left(\frac{n}{2}\right) + n \text{ (Đúng)}$$

Giả sử (1) đúng với $k = t$, nghĩa là:

$$T(t) = 2^t T\left(\frac{n}{2^t}\right) + tn$$

Ta cần chứng minh (1) cũng đúng với $k = t + 1$. Thật vậy, ta có:

$$\begin{aligned} T(t+1) &= 2 \cdot \left[2^t T\left(\frac{n}{2^t \cdot 2}\right) + \frac{n}{2}\right] + tn \\ &= 2^{t+1} \cdot T\left(\frac{n}{2^{t+1}}\right) + (t+1)\, n \text{ (đpcm)} \end{aligned}$$

$\Rightarrow$ Hệ thức (1) đúng

Thuật toán dừng lại khi $T(\frac{n}{2^k}) = 1$ hay $\frac{n}{2^k} = 1$ *(Vì $T(1) = 1$)*. Khi đó, ta có:

$$\begin{aligned} & n = 2^k \\ \Leftrightarrow\ & k = \log_2 n \end{aligned}$$

Thay k vào $T(n)$, ta có:

$$\begin{aligned} T(n) &= 2^{log_2 n} \cdot T(1) + nlog_2 n \\ &= 2^{log_2 n} + nlog_2 n \text{ (hệ thức truy hồi)} \\ &= n + nlog_2 n \end{aligned}$$

Vì ti suất tăng của $O(NlogN)$ lớn hơn so với $O(N)$ $\Rightarrow$ Độ phức tạp của thuật toán Merge Sort là: $O(NlogN)$

Chương trình được biểu diễn bằng source code python như sau:

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)>>1
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

2 Phân tích độ phức tạp thuật toán Heap Sort

Thuật toán Heap Sort có thể được phát biểu bằng lời như sau:

- Đầu tiên, ta sẽ lưu mảng vào một cấu trúc dữ liệu được gọi là *heap*
- Khi vẫn còn tồn tại các phần tử trên Heap, ta thực hiện vòng lặp: ở mỗi bước trong vòng lặp, ta sẽ lấy ra phần tử lớn nhất trên heap và cho vào mảng
- Mảng sau khi thực hiện xong sẽ là mảng đã sắp xếp

Độ phức tạp của thuật toán Heap Sort được phân tích như sau:

- Ở bước 1, để đưa các giá trị của mảng vào heap thì chúng ta cần phải thao tác với độ phức tạp là $O(n)$.
- Ở bước 2, chúng ta cần phải duyệt qua từng phần tử trên cây,ở mỗi lần lặp, chúng ta cần phải thực hiện đồng thời 2 truy vấn:
  - Xóa node root trên heap.
  - Điều chỉnh heap thành max heap.

  Biết rằng, mỗi thao tác xóa và cập nhật heap có độ phức tạp trung bình là $O(logN)$

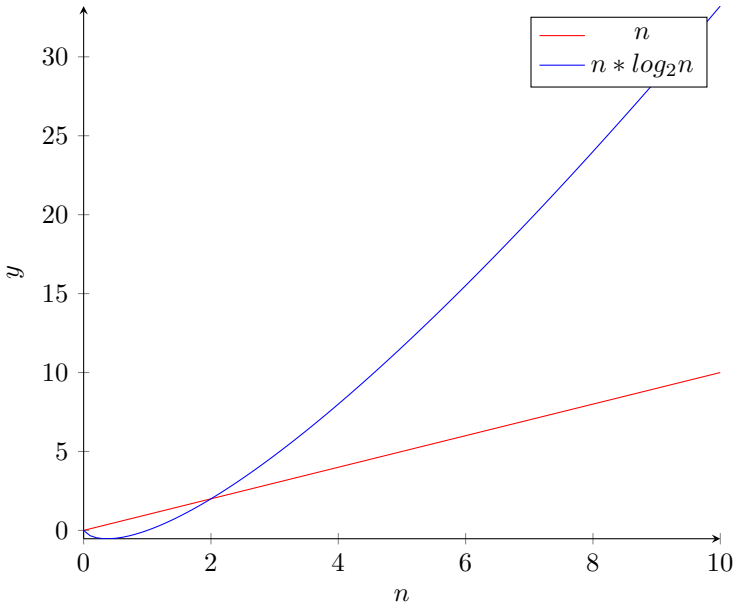
Qua đó, ta có thể tính toán độ phức tạp như sau:

$$\begin{aligned} &\textit{Độ phức tạp Heap Sort} = \\ &\quad \textit{Chi phí lặp qua từng phần tử của mảng} \\ &+ \textit{Chi phí duyệt từng phần tử trên heap} \\ &* \textit{Chi phí xóa và điều chỉnh heap} \end{aligned}$$

Nói cách khác, chúng ta có công thức:

$$O(\textit{Heap Sort}) = N + N \cdot logN \tag{2}$$

Theo quy tắc lấy max, ta có độ phức tạp của thuật toán là $O(max(N, NlogN))$, mà ta đã biết rằng tỉ suất tăng của $NlogN$ lớn hơn so với N khi ta xét $\lim_{n \to +\infty}$



Do đó *độ phức tạp trung bình của thuật toán Heap Sort* là: $O(NlogN)$

Thuật toán Heap Sort có thể mô tả bằng source code python như sau:

```
def heapify(arr, n, i):
    largest = i
    l = i << 1 + 1
    r = i << 1 + 2

    if l < n and arr[largest] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)
    for i in range(n>>1 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```