Mục lục

1. Giải thuật sắp xếp trộn(Merge Sort)

Sắp xếp trộn là một thuật toán sắp xếp dựa trên giải thuật Divide and Conquer (Chia để trị).

Thuật toán này sẽ chia mảng thành hai nữa rồi sắp xếp trên từng nữa một. Sau đó kết hợp chúng lại với nhau thành một mảng đã được sắp xếp.

a. Ý tưởng và mô tả bài toán

-Ý tưởng của giải thuật này bắt nguồn từ việc trộn 2 danh sách đã sắp xếp thành 1 danh sách mới cũng được sắp xếp.

Giả sử có hai danh sách đã được sắp xếp a[1 ... m] và b[1 ... n].

Ta có thể trộn chúng lại thành một danh sách mới c[1 ... m+n] được sắp xếp theo cách sau:

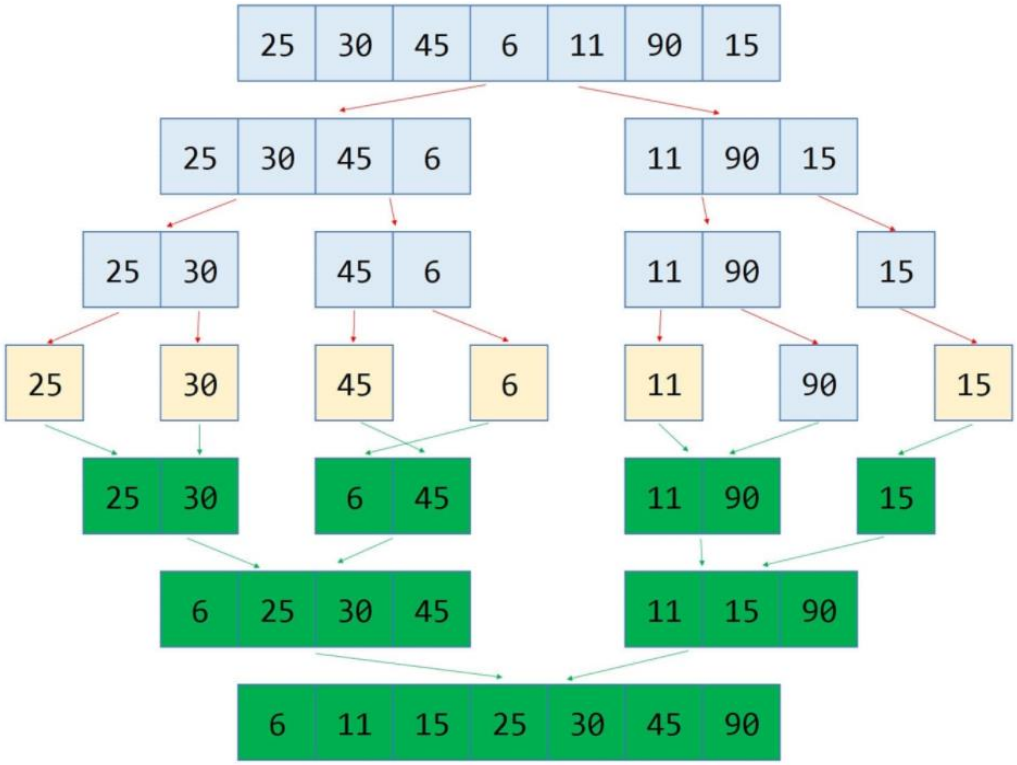
So sánh hai phần tử đứng đầu của hai danh sách, lấy phần tử nhỏ hơn cho vào danh sách mới. Tiếp tục như vậy cho tới khi một trong hai danh sách là rỗng.

Khi một trong hai danh sách là rỗng ta lấy phần còn lại của danh sách kia cho vào cuối danh sách mới.

-Ý tưởng để áp dụng cho một Merge Sort được mô tả ngắn gọn như trong code giả bên dưới:

```
// mergeSort(arr[], l,  r)
// if r > l
//      1. Tìm chỉ số nằm giữa mảng để chia mảng thành 2 nửa:
//              middle m = (l+r)/2
//      2. Gọi đệ quy hàm mergeSort cho nửa đầu tiên:
//              mergeSort(arr, l, m)
//      3. Gọi đệ quy hàm mergeSort cho nửa thứ hai:
//              mergeSort(arr, m+1, r)
//      4. Gộp 2 nửa mảng đã sắp xếp ở (2) và (3):
//              merge(arr, 1, m, r)
```

Đây là sơ đồ minh họa tiến trình từng bước của thuật toán Merge sort áp dụng cho mảng {25, 30, 45, 6, 11, 90, 15}



Nếu nhìn kỹ hơn vào sơ đồ này, chúng ta có thể thấy:

+ Mảng ban đầu được lặp lại hành động chia cho tới khi kích thước các mảng sau chia là 1.

+ Khi kích thước các mảng con là 1, tiến trình gộp sẽ bắt đầu.

+Thực hiện gộp lại các mảng này cho tới khi hoàn thành và chỉ

còn một mảng đã sắp xếp.

b. Tư tưởng giải bài toán

Trong thuật toán sẽ có hai bước đó là chia phần tử và gộp phần tử (kèm theo sắp xếp). Cụ thể trong thuật toán mình viết hàm mergeSort() để chia phần tử và hàm merge() để gộp, sắp xếp phần tử.

+ Sử dụng đệ qui để chia list thành hai nửa cho đến khi không chia thêm được nữa (hàm mergeSorrt()).

+ Tạo hai mảng tạm thời để chứa các phần tử sau khi chia, cùng với đó là hai mảng con L và R.

+ Trường hợp chỉ có một phần tử thì xem như đã được sắp xếp.

+ Sau khi chia xong, sẽ gộp các phần tử ở mảng con R và L vào mảng chính arr.

+ Kết hợp các list nhỏ đã sắp xếp với nhau thành một list mới. Sau khi kết hợp tiến hành sắp xếp chúng để tiếp tục cho lần kết hợp kế tiếp

c. Triển khai thuật tuấn bằng ngôn ngữ Python

Như bài toán chia sẻ trên, chung ta đã có ý tưởng và giải thuật rồi. Bây giờ hãy cùng mình triển khai thuật toán Merge sort này trong Python xem như thế nào nhé.

sắp xếp trộn.py

```
def mergeSort(myList):
    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]
        mergeSort(left)
        mergeSort(right)
        i = 0
        j = 0
        k = 0

        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
              myList[k] = left[i]
              i += 1
            else:
                myList[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            myList[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            myList[k]=right[j]
            j += 1
            k += 1

myList = [5,2,3,1,7,6,4,8,9]
mergeSort(myList)
print(myList)
```

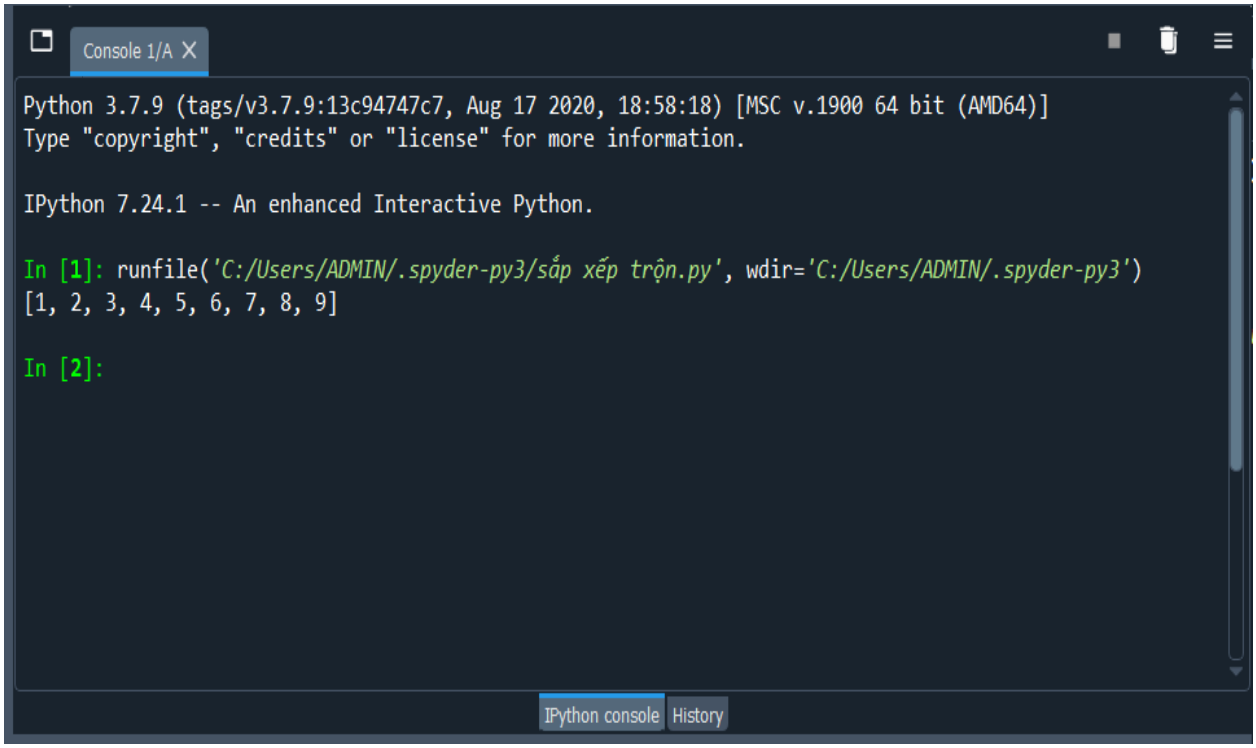
-Sau khi chạy chương trình ta nhận được kết quả:

+Mảng ban đầu:

5,2,3,1,7,6,4,8,9

+Mảng sau khi chạy chương trình:

1,2,3,4,5,6,7,8,9



d. Kết luận

Merge Sort là một trong những thuật toán sắp xếp nhanh được sử dụng rộng rãi.

Giả sử có một máy tính giải được 109 bài toán trong 1 giây:

+Trong trường hợp đó, để sắp xếp một dãy số gồm 106 số thì Merge Sort tốn chưa tới 1 giây

+Còn sử dụng thuật toán Bubble Sort thì tốn tới 1000 giây.

Ngoài ra, Merge Sort còn có tốc độ ổn định.

Vẫn là một máy tính giải được 109 bài toán trong 1 giây:

+Trong khi đó nếu ta dùng Quick Sort sắp xếp 1 dãy chứa 106 số thì vẫn có trường hợp ta phải chờ 1000 giây để sắp xếp xong

+Còn Merge Sort thì luôn luôn chưa tới 1 giây.

Như bạn thấy đó, sử dụng đúng thuật toán mang lại hiệu quả cực kỳ lớn.

Không chỉ mang lại tốc độ cực kỳ tốt, Merge Sort còn giữ được thứ tự của các phần tử bằng nhau sau khi sắp xếp.

Chẳng hạn như khi ta cần sắp xếp một mảng gồm các điểm theo trong hệ tọa độ Oxy theo chiều tăng dần của hoành độ x, nếu hoành độ bằng nhau thì sắp tăng dần theo tung độ y.

Trong trường hợp đó, ta chỉ cần dùng Merge Sort sắp xếp tung độ sau đó lại dùng Merge Sort sắp xếp hoành độ.

Thuật toán Merge sort là một giải thuật sắp xếp mà có thời gian thực hiện là O(NlogN) trong mọi trường hợp.

Chính vì thế, với dữ liệu lớn và cần ít thao tác sắp xếp thì Merge Sort sẽ tối ưu hơn Quick sort. Nó chỉ có 1 nhược điểm đó là code hơi khó cài đặt.

2. Thuật Toán Sắp Xếp Vun Đống (Heap Sort)

Sắp xếp vun đống (Heap Sort) là một kỹ thuật sắp xếp phân loại dựa trên một cấu trúc dữ liệu được gọi là đống nhị phân (binary heap), gọi đơn giản là đống. Nó tương tự như thuật toán Sắp xếp chọn (Selection Sort) nơi phần tử lớn nhất sẽ được xếp vào cuối danh sách.

a. Khái niệm "Đống"

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khóa sao cho khóa ở nút cha bao giờ cũng lớn hơn khóa ở nút con của nó.

Đống được lữu trữ trong máy bởi một vector K mà K[i] thì lưu trữ giá trị ở nút thứ i trên cây nhị phân hoàn chỉnh (Theo cách đánh số thứ tự khi lưu trữ kế tiếp).

Một cây nhị phân, được gọi là đống cực đại nếu khóa của mọi nút không nhỏ hơn khóa các con của nó.

b. Phép tạo đống

Xét một cây nhị phân hoàn chỉnh có n nút, mà mỗi nút đã được gán một giá trị khóa.

- Nếu mỗi một cây nhị phân hoàn chỉnh đã là đống thì các cây con của các nút (nếu có) cũng là đống.
- Trên cây nhị phân hoàn chỉnh có n nút thì chỉ có n/2 nút được gọi là cha.
- Một nút là cũng có thể được coi là đống

Từ đó ta thấy có thể thực hiện tạo đống từ đáy lên.

c. Sắp xếp kiểu vun đống

Sắp xếp kiểu vun đống là một cải tiến của một phương pháp sắp xếp kiểu lựa chọn. Tuy nhiên để chọn ra số lớn nhất người ta đã dựa vào cấu trúc đống và sắp xếp theo thứ tự tăng dần của các giá trị khóa là số, như ta đã quy ước, thì khóa lớn nhất sẽ được sắp xếp vào cuối dãy, nghĩa là nó đang được đổi chỗ với khóa ở đáy đống, và sau phép đổi chỗ này một khóa trong dãy của nó đã đi vào đúng vị trí sắp xếp. Nếu không kể tới khóa này thì phần còn lại của dãy ứng với một khóa của cây nhị phân hoàn chỉnh, với số khóa nhỏ hơn 1 sẽ không còn là đống nữa. Ta gặp lại bài toán tạo đống mới cho cây này và lại thực hiện tiếp phép đổi chỗ giữa khóa đỉnh với khóa ở đáy đống tương tự như đã làm.

Cho tới khi cây còn một nút thì khóa đã được sắp xếp vào đúng vị trí của nó trong dãy sắp xếp.

Như vậy sắp xếp kiểu vun đống chia làm 2 giai đoạn:

- Giai đoạn tạo đống ban đầu
- Giai đoạn sắp xếp gồm 2 bước:
    - Vun đống
    - Đổi chỗ

Được thực hiện (n-1) lần.

d. Mô tả và tư tưởng giải bài toán sắp xếp vun đống

Giả sử ta muốn sắp xếp các phần tử trong mảng A[] theo thứ tự tăng dần. Ta có thể dùng max heap để làm điều này bởi dựa vào đặc tính của max heap, node gốc trong max heap là node có giá trị lớn nhất. Như vậy sau khi tìm được node lớn nhất ta có thể lưu nó lại một nơi khác, thay thế nó bằng một giá trị khác, nhỏ hơn ở trong cây và tiếp tục áp dụng hàm run_maxheap() để tìm được giá trí lớn nhất thứ 2. Cụ thể các bước như sau:

- Bước đầu tiên, ta tạo một max heap của các phần tử trong mảng A [] bởi sử dụng hàm run_maxheap ().
- Bây giờ ta thu được phần tử có giá trị lớn nhất trong mảng, chính là node gốc của heap hay phần tử A [1] trong mảng. Đổi chỗ phần tử này với phần tử cuối cùng của mảng A (vì phần tử cuối cùng thường là những phần tử có giá trị nhỏ nhất).
- Sau khi đổi chỗ, ta tiếp tục tạo max heap mới của các phần tử trong mảng trừ phần tử cuối cùng, vì phần tử này đã được sắp xếp đúng vị trí (ta sắp xếp mảng theo thứ tự tăng dần, do vậy phần tử cuối của mảng sẽ là phần tử có giá trị cao nhất) và do vậy ta giảm kích thước của heap đi 1.
- Lập lại bước 2 và bước 3 cho đến khi tất cả các phần tử trong mảng được sắp xếp đúng theo thứ tự tăng dần.

- Và ta thu được mảng đã được sắp xếp.

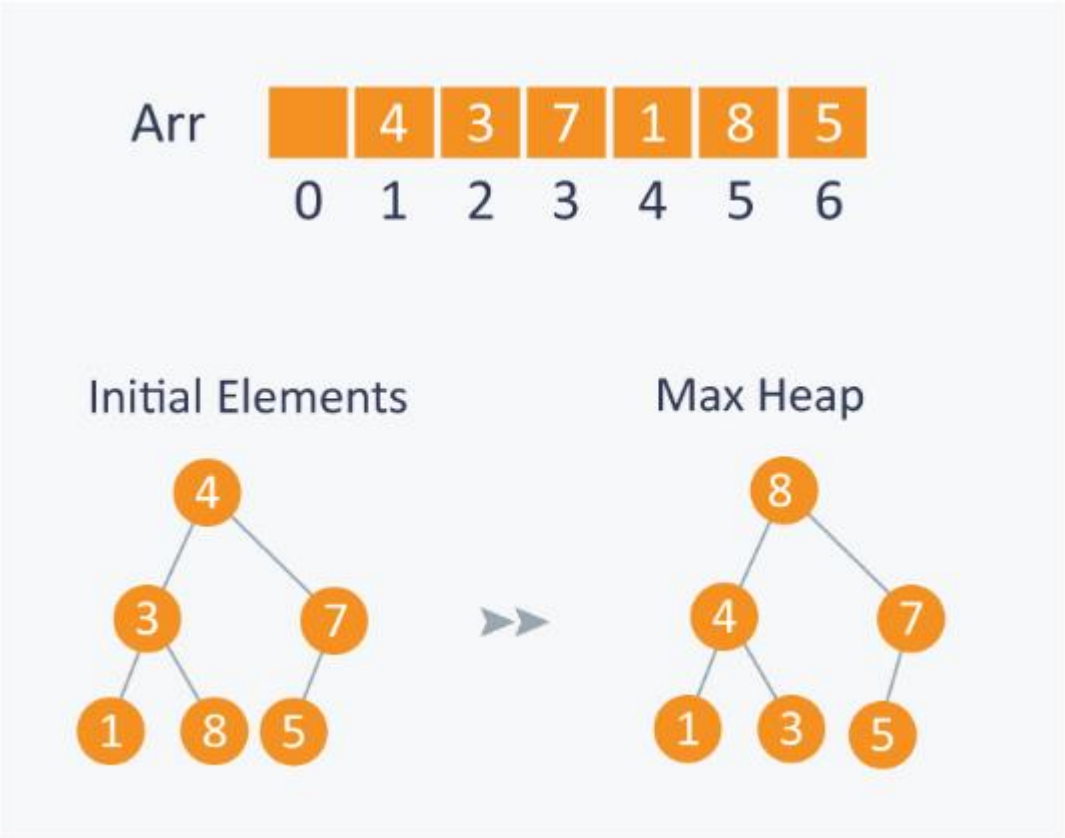
Gọi hàm thực hiện sắp xếp dựa vào max heap là max_heap_sort(), tuân theo các bước đã để cập ở trên, nội dung của hàm max_heap_sort() như sau:

```
void max_heap_sort(int A[ ])
{
    int heap_size = N;
    run_maxheap(A);
    for(int i = N; i>=2 ; i-- )
    {
        swap(A[ 1 ], A[ i ]);
        max_heap(A, 1, --heap_size);
    }
}
```

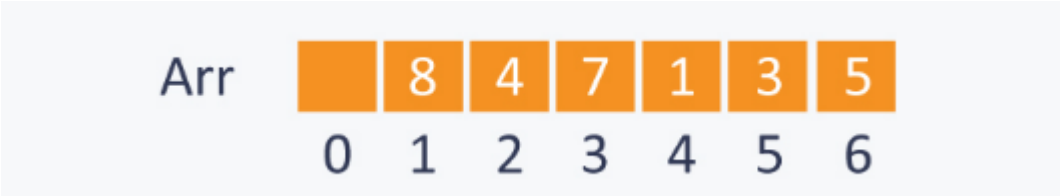
Như ta đã biết, độ phức tạp của hàm max_heap() là O(logN), run_maxheap() là O(N) và chúng ta chạy hàm max_heap() N-1 lần trong hàm max_heap_sort().

do vậy độ phức tạp của hàm max_heap_sort() sẽ bằng O(NlogN).

Trong hình bên dưới, Ta có mảng A với 6 phần tử chưa được sắp xếp. Ta thực hiện tạo max heap cho mảng này:



Sau khi tạo max heap, các phần tử trong mảng sẽ như sau:



Và các bước xử lý cần có tiếp theo để thu được mảng sắp xếp là:

Bước 1: 8 được đổi chỗ cho 5.

Bước 2: 8 được bỏ ra khỏi heap vì nó đã được sắp xếp đúng vị trí.

Bước 3: Max heap mới được tạo và 7 được đổi chỗ cho 3.

Bước 4: 7 được bỏ ra khỏi heap vì nó đã được sắp xếp đúng vị trí.

Bước 5: Max heap mới được tạo với 4 phần tử còn lại. 5 được đổi chỗ cho 1.

Bước 6: 5 được bỏ ra khỏi heap vì nó đã được sắp xếp đúng vị trí.

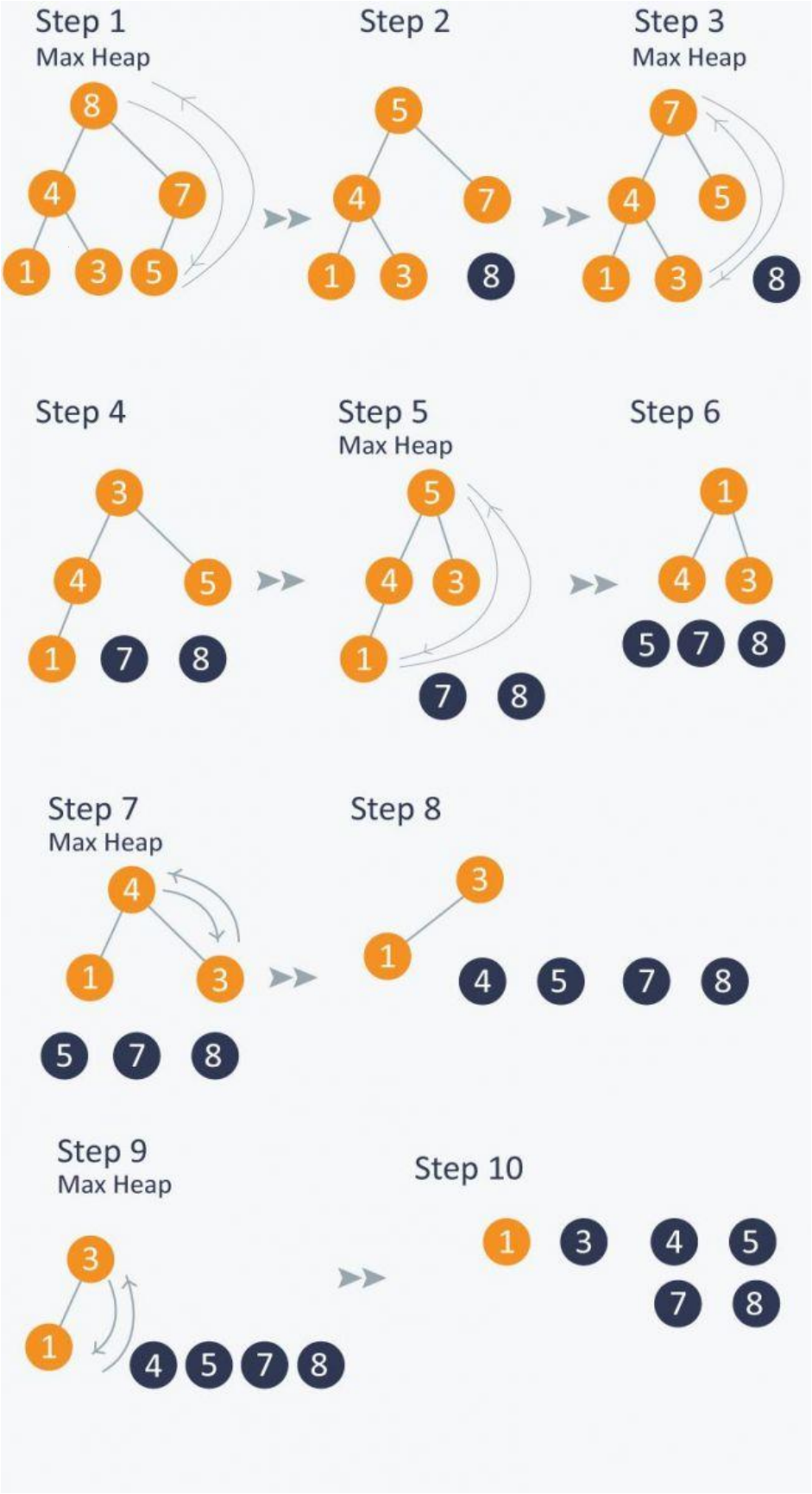
Bước 7: Max heap mới được tạo với 3 phần tử còn lại. 4 được đổi chỗ cho 3.

Bước 8: 4 được bỏ ra khỏi heap vì nó đã được sắp xếp đúng vị trí.

Bước 9: Max heap mới được tạo với 2 phần tử còn lại. 3 được đổi chỗ cho 1.

Bước 10: 3 được bỏ ra khỏi heap vì nó đã được sắp xếp đúng vị trí.

Bước 11: heap còn lại duy nhất một phần từ, ta bỏ nó ra khỏi heap và thu được mảng đã được sắp xếp theo thứ tự tăng dần.



Hình trên đã mô tả toàn bộ hoạt động của thuật toán heap sort dùng max heap. Ta có nhiều cách để thực hiện heap sort, ta có thể dùng cách trên hoặc duyệt max heap đã được tạo ra và sắp xếp lại nó theo mỗi level dùng cách duyệt theo

chiều rộng. Tuy nhiên cách trên là cách hiệu quả, dễ cài đặt và tối ưu về mặt hiệu năng cửa chương trình.

e. Code bài toán sắp xếp vun đống

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jul 11 10:12:00 2021

@author: 84969
"""

def heapify(arr, n, i):
    largest = i # set largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if (l < n and arr[i] < arr[l]):
        largest = l

    # See if right child of root exists and is
    # greater than root
    if (r < n and arr[largest] < arr[r]):
        largest = r

    # Change root, if needed
    if (largest != i):
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a heap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
```

```
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver code to test above
arr = [4, 10, 3, 5, 1]
heapSort(arr)
print ("Sorted array is")
print (arr)
```

3. Quick sort

Quick sort (sắp xếp nhanh) cũng là một **thuật toán chia để trị** giống như sắp xếp trộn. Mặc dù phức tạp hơn một chút nhưng theo các triển khai tiêu chuẩn hầu hết nó thực hiện nhanh hơn đáng kể so với sắp xếp trộn và hiếm khi đạt đến độ phức tạp trong trường hợp xấu nhất là O (n^2). Sắp xếp nhanh có 3 bước chính:

(1) Đầu tiên chúng ta chọn một phần tử mà chúng ta sẽ gọi là pivot từ mảng.

(2) Di chuyển tất cả các phần tử nhỏ hơn pivot quay sang trái của pivot; di chuyển tất cả các phần tử lớn hơn pivot sang bên phải của pivot. Đây được gọi là hoạt động phân đoạn (partition operation).

(3) Áp dụng đệ quy 2 bước trên riêng biệt cho từng mảng con của các phần tử có giá trị nhỏ hơn và lớn hơn pivot cuối cùng.

```
def partition(array, begin, end):
    pivot_idx = begin
    for i in xrange(begin+1, end+1):
        if array[i] <= array[begin]:
            pivot_idx += 1
            array[i], array[pivot_idx] = array[pivot_idx], array[i]
    array[pivot_idx], array[begin] = array[begin], array[pivot_idx]
    return pivot_idx

def quick_sort_recursion(array, begin, end):
    if begin >= end:
        return
    pivot_idx = partition(array, begin, end)
    quick_sort_recursion(array, begin, pivot_idx-1)
    quick_sort_recursion(array, pivot_idx+1, end)

def quick_sort(array, begin=0, end=None):
    if end is None:
```

```
    end = len(array) - 1

  return quick_sort_recursion(array, begin, end)
```

4. Insertion sort

Insertion sort (Sắp xếp chèn) vừa nhanh hơn và được cho là đơn giản hơn cả sắp xếp nổi bọt và sắp xếp chọn. Thật hài hước, đó là cách nhiều người sắp xếp các quân bài của họ khi chơi một trò chơi bài. Trên mỗi lần lặp vòng lặp, sắp xếp chèn loại bỏ một phần tử khỏi mảng. Sau đó, nó tìm vị trí mà phần tử đó thuộc về một mảng được sắp xếp khác và chèn nó vào đó. Nó lặp lại quá trình này cho đến khi không còn yếu tố đầu vào nào.

```
def insertion_sort(arr):

  for i in range(len(arr)):
    cursor = arr[i]
    pos = i

    while pos > 0 and arr[pos - 1] > cursor:
      # Swap the number down the list
      arr[pos] = arr[pos - 1]
      pos = pos - 1
    # Break and do the final swap
    arr[pos] = cursor

  return arr
```

5. Selection sort

Selection sort (Sắp xếp chọn) cũng khá đơn giản nhưng thường hoạt động tốt hơn bubble sort. Nếu bạn đang chọn giữa hai thuật toán thì tốt nhất nên chọn sắp xếp chọn. Với selection sort, chúng ta chia danh sách / mảng đầu vào của thành hai phần: danh sách con các mục đã được sắp xếp và danh sách con các mục còn lại sẽ được sắp xếp tạo nên phần còn lại của danh sách. Đầu tiên chúng ta tìm phần tử nhỏ nhất trong danh sách con chưa được sắp xếp và đặt nó ở cuối danh sách con đã sắp xếp. Theo cách đó, chúng ta liên tục lấy phần tử chưa được sắp xếp nhỏ nhất và đặt nó theo thứ tự được sắp xếp trong danh sách con được sắp xếp. Quá trình này tiếp tục lặp đi lặp lại cho đến khi danh sách được sắp xếp đầy đủ.

```
def selection_sort(arr):
    for i in range(len(arr)):
        minimum = i

        for j in range(i + 1, len(arr)):
            # Select the smallest value
            if arr[j] < arr[minimum]:
                minimum = j

        # Place it at the front of the
        # sorted end of the array
        arr[minimum], arr[i] = arr[i], arr[minimum]

    return arr
```

6. Bubble Sort

Bubble sort (sắp xếp nổi bọt) là cách thường được dạy trong các lớp nhập môn khoa học máy tính vì nó thể hiện rõ ràng cách sắp xếp hoạt động đồng thời là thuật toán đơn giản và dễ hiểu. Bubble sort duyệt qua danh sách và so sánh các cặp phần tử liền kề. Các phần tử được hoán đổi nếu chúng không đúng thứ tự. Việc chuyển qua phần chưa được sắp xếp của danh sách được lặp lại cho đến khi danh sách được sắp xếp. Vì bubble sort lặp đi lặp lại qua phần không được sắp xếp của danh sách, nên nó có độ phức tạp trong trường hợp xấu nhất là O (n^2).

```
def bubble_sort(arr):
    def swap(i, j):
        arr[i], arr[j] = arr[j], arr[i]

    n = len(arr)
    swapped = True

    x = -1
    while swapped:
        swapped = False
        x = x + 1
        for i in range(1, n-x):
            if arr[i - 1] > arr[i]:
                swap(i - 1, i)
                swapped = True

    return arr
```

7. Danh sách liên kết

Danh sách liên kết đơn(Single linked list) là ví dụ tốt nhất và đơn giản nhất về cấu trúc dữ liệu động sử dụng con trỏ để cài đặt. Do đó, kiến thức con trỏ là rất quan trọng để hiểu cách danh sách liên kết hoạt động, vì vậy nếu bạn chưa có kiến thức về con trỏ thì bạn nên học về con trỏ trước. Bạn cũng cần hiểu một chút về cấp phát bộ nhớ động. Để đơn giản và dễ hiểu, phần nội dung cài đặt danh sách liên kết của bài viết này sẽ chỉ trình bày về danh sách liên kết đơn

Nội dung	Mảng	Danh sách liên kết
Kích thước	Kích thước cố định Cần chỉ rõ kích thước trong khi khai báo	*Kích thước thay đổi trong quá trình thêm/ xóa phần tử* *Kích thước tối đa phụ thuộc vào bộ nhớ*
Cấp phát bộ nhớ	Tĩnh: Bộ nhớ được cấp phát trong quá trình biên dịch	*Động: Bộ nhớ được cấp phát trong quá trình chạy*
Thứ tự & sắp xếp	Được lưu trữ trên một dãy ô nhớ liên tục	*Được lưu trữ trên các ô nhớ ngẫu nhiên*
Truy cập	*Truy cập tới phần tử ngẫu nhiên trực tiếp bằng cách sử dụng chỉ số mảng: O(1)*	Truy cập tới phần tử ngẫu nhiên cần phải duyệt từ đầu/cuối đến phần tử đó: O(n)
Tìm kiếm	*Tìm kiếm tuyến tính hoặc tìm kiếm nhị phân*	Chỉ có thể tìm kiếm tuyến tính

Lợi ích của Danh sách được Liên kết

Danh sách Liên kết cho phép chúng tôi linh hoạt trong **việc phân bổ bộ nhớ** , vì nó không yêu cầu bạn có không gian trống dài để lưu trữ dữ liệu trong bộ nhớ. Ngoài ra, nếu bạn muốn thêm một phần tử mới vào một mảng nằm ngoài phạm vi chỉ mục, mảng cần tạo lại một mảng lớn hơn để chứa tất cả. Trong danh sách được liên kết, bạn có thể chỉ cần thêm một phần tử bổ sung với tham chiếu của phần tử tiếp theo.

Hạn chế của danh sách được liên kết

Theo tôi thấy, có hai nhược điểm của danh sách liên kết. Một, **chỉ có một điểm vào** được gọi là **head** Giả sử bạn muốn lấy dữ liệu từ hộp thứ ba, trước tiên bạn sẽ làm gì để kiểm tra xem trong hộp thứ ba có gì? Câu trả lời là hãy mở hộp đầu tiên để lấy hộp tiếp theo! Sau đó, bạn sẽ cần phải chọn hộp thứ hai để lấy địa chỉ

cho hộp thứ ba và bây giờ bạn thấy có dữ liệu 3 trong hộp thứ ba. Hai, mỗi hộp yêu cầu **thêm không gian bộ nhớ** cho một **con trỏ** , là địa chỉ cho hộp tiếp theo.

Hãy truyền tải ý tưởng trực quan vào các dòng mã.

Danh sách liên kết chứa các nút của chúng ta sẽ được triển khai như sau, phần **đầu** đại diện cho phần tử đầu tiên trong danh sách được liên kết và null (hoặc Không có) theo mặc định. Tuy nhiên, hiện tại chúng tôi KHÔNG sử dụng danh sách này để đơn giản hóa.

```
class LinkedList:
    def __init__(self):
        self.head = None
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
# Create nodes(post boxes)
nodeOne = new Node(1)
nodeTwo = new Node(2)
nodeThree = new Node(3)
# Link them with a pointer
nodeOne.next = nodeTwo
nodeTwo.next = nodeThree
```

Trong cấu trúc dữ liệu, dường như luôn có **các phương pháp** để **thêm** và **xóa** dữ liệu. Cũng phải có các phương pháp, nhưng tôi sẽ đề cập đến những phương pháp rất thường được sử dụng.

append ()

Độ phức tạp thời gian: O (N)

append () là một phương thức thêm dữ liệu vào **cuối** danh sách được liên kết và độ phức tạp về thời gian của nó là **O (N)** vì nó cần phải đi qua tất cả các cách đến cuối cùng.

```
class LinkedList:
    def __init__(self):
        self.head = None
```

```
    def push(self, data):
        new_node = Node(data)
if self.head is None:
            self.head = new_node
            return
current = self.head
while current_node.next:
            current = current.next
current.next = new_node
```

1. Tạo một nút mới bằng cách sử dụng hàm tạo nút có chứa các vị trí cho dữ liệu và tham chiếu nút tiếp theo.
2. Kiểm tra xem có nút đầu hay không, không có đầu nghĩa là danh sách trống nên new_node trở thành nút đầu trong trường hợp này.
3. Nếu không, chúng ta cần lặp lại danh sách của mình cho đến cuối, vì chúng ta luôn cần bắt đầu từ nút đầu trong danh sách.
4. Để nút cuối cùng trong danh sách trỏ tới new_node.

Độ phức tạp thời gian: O (N)

Có hai cách để xóa dữ liệu, tôi sẽ hướng dẫn bạn cách xóa dữ liệu chính xác mà tôi muốn xóa. Trong trường hợp này, **chúng tôi sẽ xóa 2 khỏi danh sách liên kết của mình** , sau đó chúng tôi chỉ cần lặp lại một danh sách cho đến khi chúng tôi tìm thấy cùng một khóa từ danh sách. nó hoạt động gần giống như trong danh sách liên kết lặp, ngoại trừ một điều, tham chiếu cho **trước đó** , vì không có cách nào để lấy tham chiếu trước đó, chúng tôi sẽ tạm thời chứa nó trong một biến.

Trước khi lặp, chúng ta cần kiểm tra xem **phần tử đầu tiên có phải là khóa mà chúng ta đang theo dõi hay** không, nếu có, chúng ta chỉ có thể **thay đổi đầu thành phần tử tiếp theo** và dừng phương thức. Trong trường hợp này, độ phức tạp về thời gian là thời gian không đổi - O (1)

```
if current is not None and current.data == key:
  self.head = current.next
  return
```

8. Thuật toán tìm kiếm trong Python

Thuật toán tìm kiếm tuyến tính:

Trong khoa học máy tính, tìm kiếm tuyến tính hoặc tìm kiếm tuần tự là một phương pháp để tìm một phần tử trong danh sách. Nó tuần tự kiểm tra từng phần tử của danh sách cho đến khi tìm thấy kết quả khớp hoặc toàn bộ danh sách đã được tìm kiếm.

Trong tìm kiếm tuyến tính, bạn lặp qua tất cả các phần tử trong danh sách một cách đơn giản cho đến khi bạn gặp giá trị mong muốn mà bạn đang tìm kiếm. Việc triển khai mã cho tìm kiếm tuyến tính khá đơn giản và sẽ được thảo luận trong phần sau của bài viết này.

Tìm kiếm tuyến tính chạy vào thời điểm tuyến tính xấu nhất và thực hiện nhiều nhất n phép so sánh, trong đó n là độ dài của danh sách. Nếu mỗi phần tử có khả năng được tìm kiếm như nhau, thì tìm kiếm tuyến tính có một trường hợp trung bình là n + 1/2 so sánh. Nhưng trường hợp trung bình có thể bị ảnh hưởng nếu xác suất tìm kiếm cho mỗi phần tử khác nhau.

Trực giác khôn ngoan của thuật toán tìm kiếm nhị phân:

Tìm kiếm nhị phân là một thuật toán hiệu quả để tìm một mục từ danh sách các mục đã được sắp xếp. Nó hoạt động bằng cách liên tục chia đôi phần danh sách có thể chứa mục cho đến khi bạn thu hẹp các vị trí có thể chỉ còn một. Hãy để chúng tôi hiểu cách triển khai từng bước của thuật toán này.

1. Sắp xếp danh sách hoặc mảng bạn có theo thứ tự tăng dần.
2. Phần tử bắt đầu được gọi là 'L' và phần tử cuối cùng được gọi là 'R'. Sử dụng các giá trị này, chúng tôi cố gắng thu hẹp phần tử ở giữa bằng công thức **(L + R) // 2** . Hoạt động này về cơ bản thực hiện một chức năng sàn để đạt được phần tử giữa mong muốn. Một cách tiếp cận thay thế là sử dụng hàm ceil, nhưng chúng tôi sẽ đề cập đến phương pháp này trong bài viết sau hôm nay.
3. Bước tiếp theo là kiểm tra xem giá trị của phần tử ở giữa có tương đương với giá trị mong muốn hay không. Nếu có, thì tìm kiếm thành công và có thể kết thúc.
4. Nếu giá trị mong muốn nhỏ hơn phần tử ở giữa, tất cả các giá trị từ phần tử ở giữa đến giá trị 'R' đều bị loại bỏ. Và sự lặp lại của bước 2 và 3 diễn ra.
5. Nếu giá trị mong muốn lớn hơn phần tử giữa, tất cả các giá trị từ phần tử giữa đến giá trị 'L' đều bị loại bỏ. Và sự lặp lại của bước 2 và 3 diễn ra.

Chúng ta sẽ bắt đầu với việc triển khai thuật toán tìm kiếm tuyến tính, thuật toán này khá đơn giản và có thể được thực hiện trong một vài dòng mã. Khối mã sau đây có thể được sử dụng để thực hiện tìm kiếm tuyến tính.

```
# Linear Search

data = [1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71]

for i in range(len(data)):
    if data[i] == 7:
        print(i)
```

Tiếp theo, chúng ta sẽ xem xét việc thực hiện thuật toán tìm kiếm nhị phân. Có nhiều phương pháp thực hiện thuật toán này, như cách lặp lại hoặc đệ quy. Trong bài viết này, chúng tôi sẽ tập trung vào một phương pháp đơn giản để thực hiện tác vụ này.

Vấn đề tương tự được giải thích trong trực giác được giải quyết bằng thuật toán tìm kiếm nhị phân và điều này có thể được thực hiện từ khối mã được hiển thị bên dưới.

```
# Binary Search

def binary_search(data, elem):

    low = 0
    high = len(data) - 1

    while low <= high:

        middle = (low + high)//2

        if data[middle] == elem:
            return middle
        elif data[middle] > elem:
            high = middle - 1
        else:
            low = middle + 1

    return -1

data = [1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71]
elem = 7

binary_search(data, elem)
```

Thời gian phức tạp

Tình huống Trường hợp Tốt nhất = **O (1)**

Kịch bản trường hợp trung bình = **O (log n)**

Kịch bản trường hợp tồi tệ nhất = **O (log n)**

Tìm kiếm nhị phân chạy theo thời gian logarit trong trường hợp xấu nhất, thực hiện so sánh O (log n), trong đó n là số phần tử trong mảng. Tìm kiếm nhị phân nhanh hơn tìm kiếm tuyến tính ngoại trừ các mảng nhỏ. Tuy nhiên, mảng phải được sắp xếp trước để có thể áp dụng tìm kiếm nhị phân.

Có những cấu trúc dữ liệu chuyên biệt được thiết kế để tìm kiếm nhanh, chẳng hạn như bảng băm, có thể được tìm kiếm hiệu quả hơn so với tìm kiếm nhị phân. Tuy nhiên, tìm kiếm nhị phân có thể được sử dụng để giải quyết nhiều vấn đề hơn, chẳng hạn như tìm phần tử nhỏ nhất hoặc lớn nhất tiếp theo trong mảng so với mục tiêu ngay cả khi nó không có trong mảng.

9. Kết luận

Điểm đặc biệt của ngôn ngữ Lập trình Python có cú pháp khá dễ hiểu, dễ đọc và dễ học. Trong việc phát triển ứng dụng thì ngôn ngữ này cũng rất linh hoạt. Python hỗ trợ mẫu đa lập trình, bao gồm lập trình hướng đối tượng, lập trình hàm và mệnh lệnh hoặc là các phong cách lập trình theo thủ tục.

Python là ngôn ngữ lập trình động nên không cần sử dụng các kiểu dữ liệu khai báo.

Ưu điểm

- Đơn giản: Là một ngôn ngữ có hình thức sáng sủa, cấu trúc rõ ràng, cú pháp ngắn gọn giúp người lập trình dễ dàng đọc và tìm hiểu.

- Tốc độ xử lý khá nhanh, và được đánh giá nhanh hơn so với ngôn ngữ PHP. Với tốc độ xử lý cực nhanh, Python có thể tạo ra những chương trình từ những script siêu nhỏ tới những phần mềm cực lớn như Biender 3D.

- Chất lượng: Thư viện có tiêu chuẩn cao, Python có khối cơ sở dữ liệu khá lớn nhằm cung cấp giao diện cho tất cả các CSDL thương mại.

- Thuận tiện: Python được biên dịch và chạy trên tất cả các nền tảng lớn. Nó có trên tất cả các nền tảng hệ điều hành từ UNIX, MS – DOS, Mac OS, Windows và Linix và các OS khác thuộc họ Unix.

Tương thích mạnh mẽ với Unix, hardware, thirt-party software với số lượng thư viện khổng lồ (400 triệu người sử dụng)

- Mở rộng: Với tính năng này, Python cho phép người lập trình có thể thêm hoặc tùy chỉnh các công cụ nhằm tối đa hiệu quả có thể đạt được trong công việc.

- GUI Programming: Giúp cho việc thực hiện ảnh minh họa di động một cách tự nhiên và sống động hơn.

Hạn chế:

Python không có các thuộc tính như :protected,private hay public, không có vòng lặp do…while và switch….case.

Mặc dù tốc độ xử lý của Python nhanh hơn PHP nhưng không bằng Java và C++.

Không có ngôn ngữ nào là hoàn hảo và cũng không phải ngẫu nhiên mà Python được nhiều lập trình viên chọn lựa để phát triển web.

Python phù hợp để giảng dạy cho Ngành Công nghệ thông tin