BRUTE FORCE

Definition:

* Brute force là một dạng giải quyết vấn đề một cách chân thật nhất, thông thường là dựa vào vấn đề được đặt ra và định nghĩa của các khái niệm có liên quan.

Idea of althogrithm:

* “Force” bao hàm bởi định nghĩa của chiến lược là định nghĩa của máy tính chứ không phải là định nghĩa của trí tuệ.
* “Just do it” có thể là một cách khác để mô tả qui tắc của các tiếp cận brute-force. Và thông thường, chiến thuật “brute-force” là một trong những cách đơn giản nhất được áp dụng.

Example for brute-force althorgithm:

1. Selection Sort & Bubble Sort:

+ Selection Sort:

* Ý tưởng giải thuật :

Chúng ta bắt đầu thực hiện phương pháp sắp xếp chọn bằng cách quét toàn bộ danh sách được cho, sau đó tiếp tục tìm ra phần tử nhỏ nhất trong danh sách và đổi chỗ với phần tử đầu tiên trong danh sách. Kế đến, chúng ta tiếp tục quét danh sách và bắt đầu với phần tử thứ hai để tìm ra phần tử nhỏ nhất trong số n-1 phần tử và đổi chỗ với phần tử thứ 2 của danh sách. Nói tóm lại, phần tử thứ i mỗi lần quét trong danh sách, chúng ta sẽ đánh số từ 0 tới n-2, giải thuật tìm kiếm phần tử nhỏ nhất trong số n-1 phần tử cuối và đổi với Ai :

Trong vị trí cuối cùng của danh sách Phần tử n – i cuối cùng

Sau khi quét qua n – 1 lần, danh sách đã được sắp xếp.

* Mã giả :

﻿

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | SelectionSort(A[**0.**.n−**1**])  //Sắp xếp mảng cho được cho bằng phương pháp sắp xếp chọn  //Input: Một mảng A[**0.**.n−**1**] chưa được sắp xếp  //Output: Một mảng A[**0.**.n−**1**] được sắp xếp theo thứ tự tăng dần  **for** i ← **0** to n − **2** do  min ← i  **for** j ← i + **1** to n − **1** do  **if** A[j] < A[min] min ← j  swap A[i] **and** A[min] | |

* Phân tích giải thuật :

Kích thước đầu vào được cho bởi tổng số các phần tử n; phép toán cơ bản là só sánh ở dòng thứ 8 “A[j] < A[min]”. Tổng số lần thực thi chỉ phụ thuộc vào kích thước của mảng được tính bằng công thức sau:

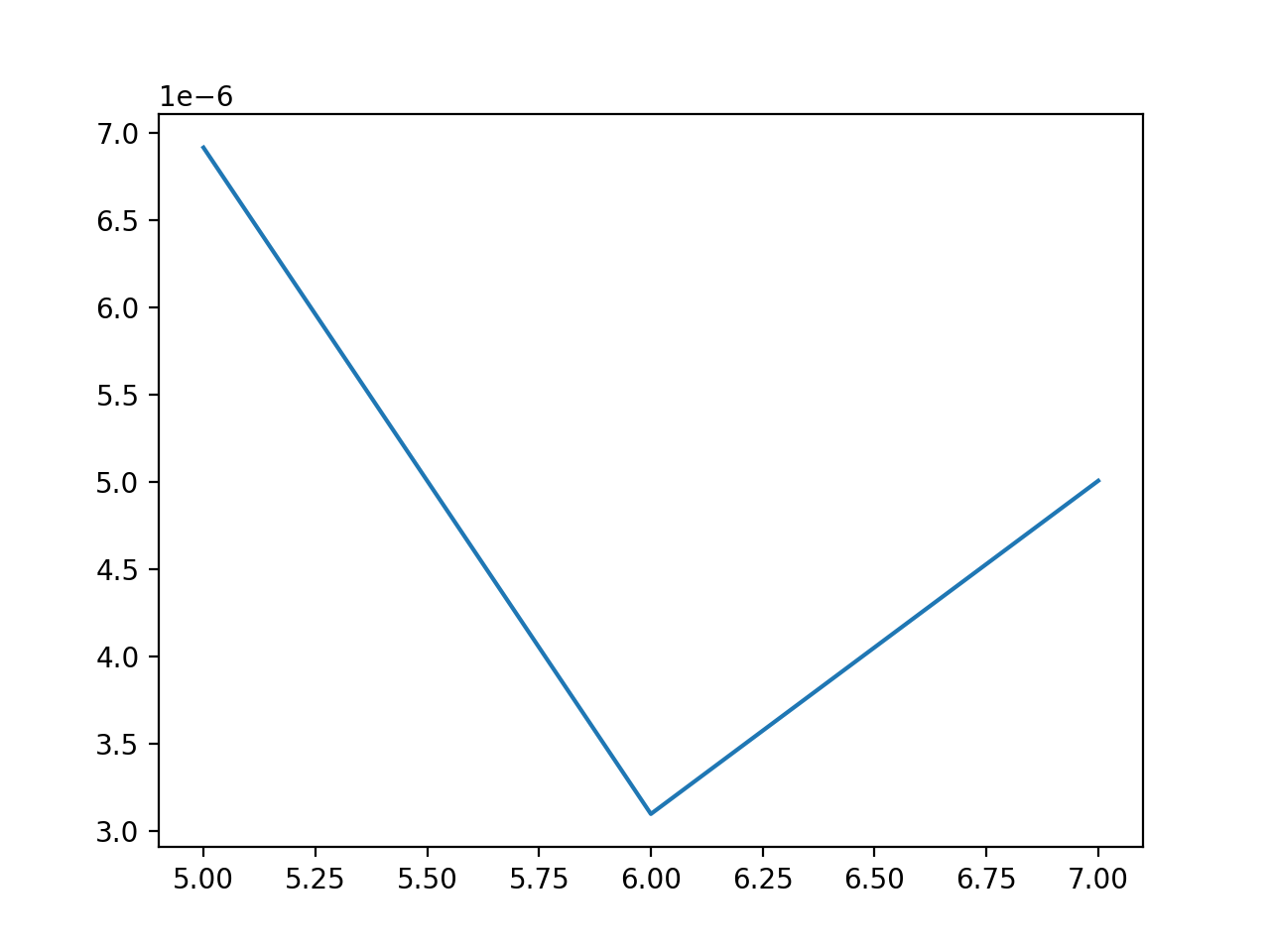
|  |
| --- |
|  |

* Triển khai với python3 :

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # Sắp xếp mảng cho được cho bằng phương pháp sắp xếp chọn  # Input: Một mảng A[0..n−1] chưa được sắp xếp  # Output: Một mảng A[0..n−1] được sắp xếp theo thứ tự tăng dần  **def** **SelectionSort**(A):  **for** i **in** range(len(A)):  min = i  **for** j **in** range(i+**1**,len(A)):  **if** A[j] < A[min]:  min = j  A[i],A[min] = A[min],A[i]  A = [**64**, **25**, **12**, **22**, **11**]  SelectionSort(A)  print(A)  #Output : [11, 12, 22, 25, 64] | |

* Demo :

|  |
| --- |
| [64, 25, 12, 22, 11]  [11, 12, 22, 25, 64]  Time A: 6.9141387939453125e-06 seconds  Input Size A: 5  [64, 25, 12, 22, 11, 5]  [64, 25, 12, 22, 11, 5]  Time B: 3.0994415283203125e-06 seconds  Input Size B: 6  [64, 25, 12, 22, 11, 5, 70]  [5, 11, 12, 22, 25, 64, 70]  Time C: 5.0067901611328125e-06 seconds  Input Size A: 7 |



+ Bubble Sort:

* Ý tưởng giải thuật :

Một ứng dụng khác của thuật toán brute-force trong việc sắp xếp là so sánh các phần tử liền kề của danh sách và đổi chỗ chúng nếu chúng không theo thứ tự. Bằng việc lập đi lập lại, chúng ta sẽ đẩy phần tử lớn nhất lên vị trí cuối cùng trong danh sách. Kế đến là thực hiện tương tự với phần tử lớn thứ 2, và cứ tiếp tục như vậy, cho đến khi sau lần thứ n - 1, danh sách được sắp xếp. Vượt qua i (0 ≤ i ≤ n - 2) của sắp xếp bong bóng có thể được biểu diễn bằng sơ đồ sau:

Trong vị trí cuối cùng của danh sách

* Mã giả :

﻿

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7 | BubbleSort(A[**0.**.n − **1**])  //Sắp xếp một mảng cho trước bằng phương pháp sắp xếp bong bóng  //Input: Một mảng A[**0.**.n−**1**] chưa được sắp xếp  //Output: Một mảng A[**0.**.n−**1**] được sắp xếp theo thứ tự tăng dần  **for** i ← **0** to n − **2** do  **for** j ← **0** to n − **2** − i do  **if** A[j + **1**] < A[j] swap A[j] **and** A[j + **1**] | |

* Phân tích giải thuật :

Kích thước đầu vào được cho bởi tổng số các phần tử n; phép toán cơ bản là só sánh ở dòng thứ 7 “A[j+1] < A[j]”. Tổng số lần thực thi chỉ phụ thuộc vào kích thước của mảng được tính bằng công thức sau:

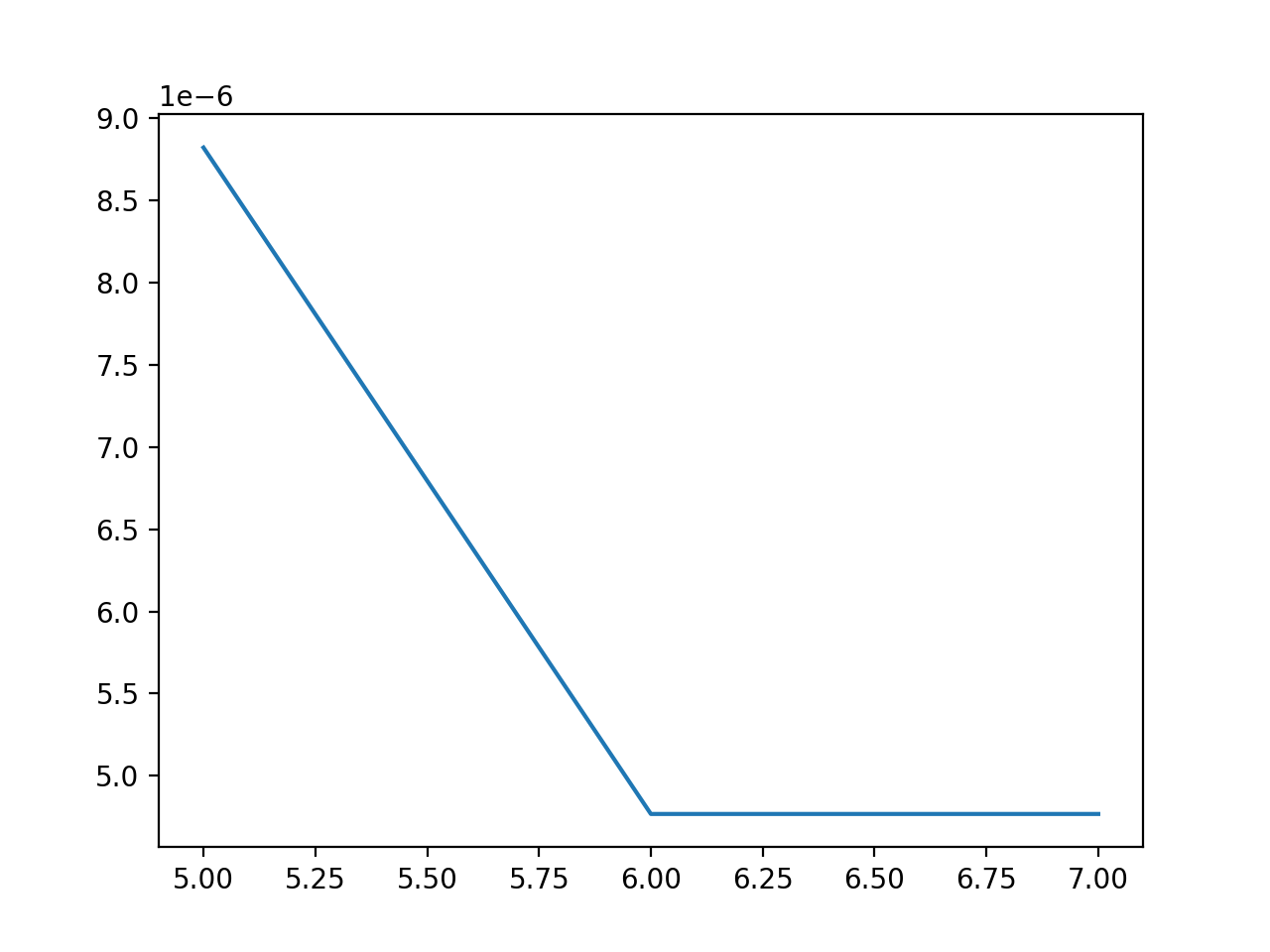
|  |
| --- |
|  |

* Triển khai với python3 :

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12 | # Sắp xếp mảng cho được cho bằng phương pháp sắp xếp bong bóng  # Input: Một mảng A[0..n−1] chưa được sắp xếp  # Output: Một mảng A[0..n−1] được sắp xếp theo thứ tự tăng dần  **def** **BubbleSort**(A):  **for** i **in** range(len(A)-**1**):  **for** j **in** range(len(A)-**1**-i):  **if** A[j+**1**] < A[j]:  A[j],A[j+**1**] = A[j+**1**],A[j]  A = [**64**, **25**, **12**, **22**, **11**]  BubbleSort(A)  print(A)  #Output : [11, 12, 22, 25, 64] | |

* Demo :

|  |
| --- |
| [64, 25, 12, 22, 11]  [11, 12, 22, 25, 64]  Time A: 8.821487426757812e-06 seconds  Input Size A: 5  [64, 25, 12, 22, 11, 5]  [64, 25, 12, 22, 11, 5]  Time B: 4.76837158203125e-06 seconds  Input Size B: 6  [64, 25, 12, 22, 11, 5, 70]  [5, 11, 12, 22, 25, 64, 70]  Time C: 4.76837158203125e-06 seconds  Input Size C: 7 |



+ Sequential Search:

* Ý tưởng giải thuật :

Thuật toán chỉ cần so sánh các phần tử liên tiếp của một danh sách nhất định với một khóa tìm kiếm nhất định cho đến khi gặp một kết quả phù hợp (tìm kiếm thành công) hoặc danh sách hết mà không tìm thấy kết quả phù hợp (tìm kiếm không thành công). Một thủ thuật bổ sung đơn giản thường được sử dụng trong việc triển khai tìm kiếm tuần tự: nếu chúng ta nối khóa tìm kim vào cuối danh sách, thì việc tìm kiếm khóa sẽ phải thành công và do đó chúng ta có thể loại bỏ hoàn toàn việc kiểm tra cuối danh sách.

* Mã giả :

﻿

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11 | SequentialSearch2(A[**0.**.n], K)  //Tìm kiếm tuần tự với khóa tìm kiếm dưới dạng chốt chặn  //Input: Một mảng A gồm n phần tử và một khóa tìm kiếm K  //Output: Chỉ số của phần tử đầu tiên trong A [**0.**.n - **1**] có giá  //trị bằng K hoặc −**1** nếu không tìm thấy phần tử đó  A[n]← K  i ← **0**  **while** A[i]!= K do  i ← i + **1**  **if** i < n **return** i  **else** **return** −**1** | |

* Triển khai với python3 :

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12 | # Sắp xếp mảng cho được cho bằng phương pháp sắp xếp bong bóng  # Input: Một mảng A[0..n−1] chưa được sắp xếp  # Output: Một mảng A[0..n−1] được sắp xếp theo thứ tự tăng dần  **def** **BubbleSort**(A):  **for** i **in** range(len(A)-**1**):  **for** j **in** range(len(A)-**1**-i):  **if** A[j+**1**] < A[j]:  A[j],A[j+**1**] = A[j+**1**],A[j]  A = [**64**, **25**, **12**, **22**, **11**]  BubbleSort(A)  print(A)  #Output : [11, 12, 22, 25, 64] | |

* Demo :

|  |
| --- |
| [64, 25, 12, 22, 11]  [11, 12, 22, 25, 64]  Time A: 8.821487426757812e-06 seconds  Input Size A: 5  [64, 25, 12, 22, 11, 5]  [64, 25, 12, 22, 11, 5]  Time B: 4.76837158203125e-06 seconds  Input Size B: 6  [64, 25, 12, 22, 11, 5, 70]  [5, 11, 12, 22, 25, 64, 70]  Time C: 4.76837158203125e-06 seconds  Input Size C: 7 |

