TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI CUỐI KÌ MÔN PHÂN TÍCH THIẾT KẾ VÀ GIẢI THUẬT**

**TÊN ĐỀ TÀI**

*Người hướng dẫn*: **NGUYỄN CHÍ THIỆN**

*Người thực hiện*: **NGUYỄN QUỐC CƯỜNG - 518H0003**

**HÀ QUÁN QUÂN – 518H0047**

Lớp **: 18H50203**

Khoá  **: 22**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI CUỐI KÌ MÔN PHÂN TÍCH THIẾT KẾ VÀ GIẢI THUẬT**

**TÊN ĐỀ TÀI**

*Người hướng dẫn*: **NGUYỄN CHÍ THIỆN**

*Người thực hiện*: **NGUYỄN QUỐC CƯỜNG - 518H0003**

**HÀ QUÁN QUÂN – 518H0047**

Lớp **: 18H50203**

Khoá  **: 22**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

# LỜI CẢM ƠN

Xin gửi lời cảm ơn chân thành đến thầy Nguyễn Chí Thiện, dù trong thời điểm vừa qua đợt dịch nên việc đến trường còn khó khăn nhưng thầy vẫn giúp đỡ tận tình để hoàn thành được bài cuối kì này.

**ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI**

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi xin cam đoan đây là sản phẩm đồ án của riêng tôi và được sự hướng dẫn của thầy Nguyễn Chí Thiện;. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

**Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình.** Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 26 tháng 11 năm 2020*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Nguyễn Quốc Cường*

*Hà Quán Quân*

# PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

## Phần xác nhận của giảng viên hướng dẫn

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

## Phần đánh giá của GV chấm bài

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

MỤC LỤC

[LỜI CẢM ƠN 3](#_Toc57059221)

[PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN 5](#_Toc57059222)

[1. Phần xác nhận của giảng viên hướng dẫn 5](#_Toc57059223)

[2. Phần đánh giá của GV chấm bài 5](#_Toc57059224)

[CHƯƠNG I: THUẬT TOÁN BRUTE-FORCE 8](#_Toc57059225)

[1. Giới thiệu về thuật toán Brute-Force: 8](#_Toc57059226)

[2. Ý thưởng của thuật toán: 8](#_Toc57059227)

[3. Các ví dụ cho thuật toán: 8](#_Toc57059228)

[1. Giải thuật Selection Sort (Sắp xếp chọn): 8](#_Toc57059229)

[2. Giải thuật Bubble Sort (Sắp xếp bong bóng): 10](#_Toc57059230)

[3. Giải thuật Sequential Search (Tìm kiếm tuần tự): 12](#_Toc57059231)

[CHƯƠNG II: THUẬT TOÁN DIVIDE AND CONQUER 15](#_Toc57059232)

[1. Giới thiệu về thuật toán Divide & Conquer: 15](#_Toc57059233)

[2. Ý thưởng của thuật toán: 15](#_Toc57059234)

[3. Các ví dụ cho thuật toán: 15](#_Toc57059235)

[1. Giải thuật Merge Sort (Sắp xếp hợp nhất): 15](#_Toc57059236)

[2. Giải thuật Quick Sort (Sắp xếp nhanh): 18](#_Toc57059237)

[3. Giải thuật Binary Tree Traversals (Duyệt cây nhị phân): 21](#_Toc57059238)

[CHƯƠNG III: THUẬT TOÁN GREEDY 24](#_Toc57059239)

[1. Giới thiệu về thuật toán Greedy: 24](#_Toc57059240)

[2. Ý thưởng của thuật toán: 24](#_Toc57059241)

[3. Các ví dụ cho thuật toán: 24](#_Toc57059242)

[1. Giải thuật Prim: 24](#_Toc57059243)

[2. Giải thuật Kruskal (Sắp xếp bong bóng): 26](#_Toc57059244)

[3. Giải thuật Dijkstra: 28](#_Toc57059245)

[CHƯƠNG IV: THUẬT TOÁN DYNAMIC PROGRAMMING 31](#_Toc57059246)

[1. Giới thiệu về thuật toán Dynamic Programming: 31](#_Toc57059247)

[2. Ý thưởng của thuật toán: 31](#_Toc57059248)

[3. Các ví dụ cho thuật toán: 31](#_Toc57059249)

[1. Giải thuật Selection Sort (Sắp xếp chọn): 31](#_Toc57059250)

[2. Giải thuật Bubble Sort (Sắp xếp bong bóng): 33](#_Toc57059251)

[3. Giải thuật Sequential Search (Tìm kiếm tuần tự): 35](#_Toc57059252)

# CHƯƠNG I: THUẬT TOÁN BRUTE-FORCE

## Giới thiệu về thuật toán Brute-Force:

* Thuật toán vét cạn (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.

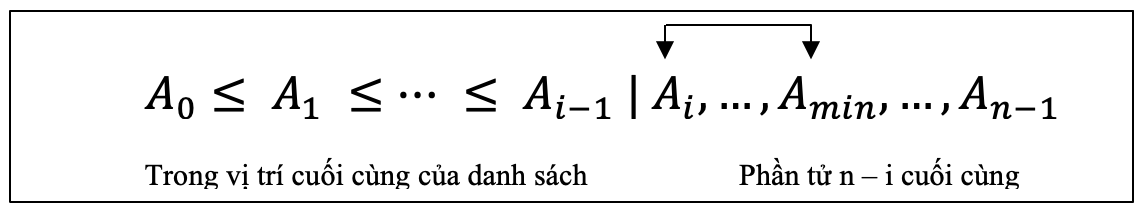
## Ý thưởng của thuật toán:

* “Đẩy tới” (force) bao hàm bởi định nghĩa của chiến lược là định nghĩa của máy tính chứ không phải là định nghĩa của trí tuệ.
* “Hãy làm đi” (just do it) có thể là một cách khác để mô tả qui tắc của các tiếp cận vét cạn. Và thông thường, chiến thuật vét cạn là một trong những cách đơn giản nhất được áp dụng.

## Các ví dụ cho thuật toán:

### Giải thuật Selection Sort (Sắp xếp chọn):

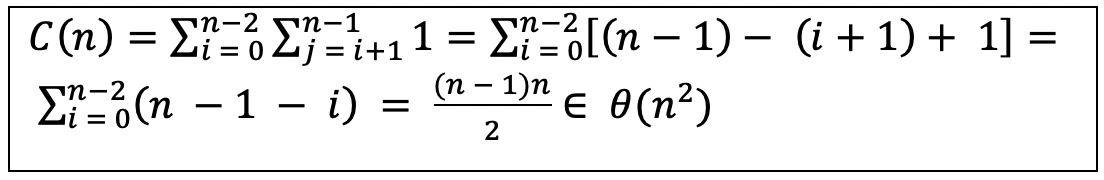
#### Ý tưởng giải thuật:

* 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:
* Sau khi quét n-1 lần, danh sách đã được sắp xếp.

#### Mã giả:

|  |
| --- |
| SelectionSort(A[**0.**.n−**1**])  # Sorts a given array by selection sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order  #import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Sorts a given array by selection sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing orderimport time  def SelectionSort(*A*):  n = len(*A*)  for i in range(n-1):  min = i  for j in range(i+1,n):  if *A*[j] < *A*[min]:  min = j  *A*[i],*A*[min] = *A*[min],*A*[i]  A = [4,7,3,2,7,5,1]  SelectionSort(A)  print(A)  # [1, 2, 3, 4, 5, 7, 7] |

Code 1:Selection Sort với python 3

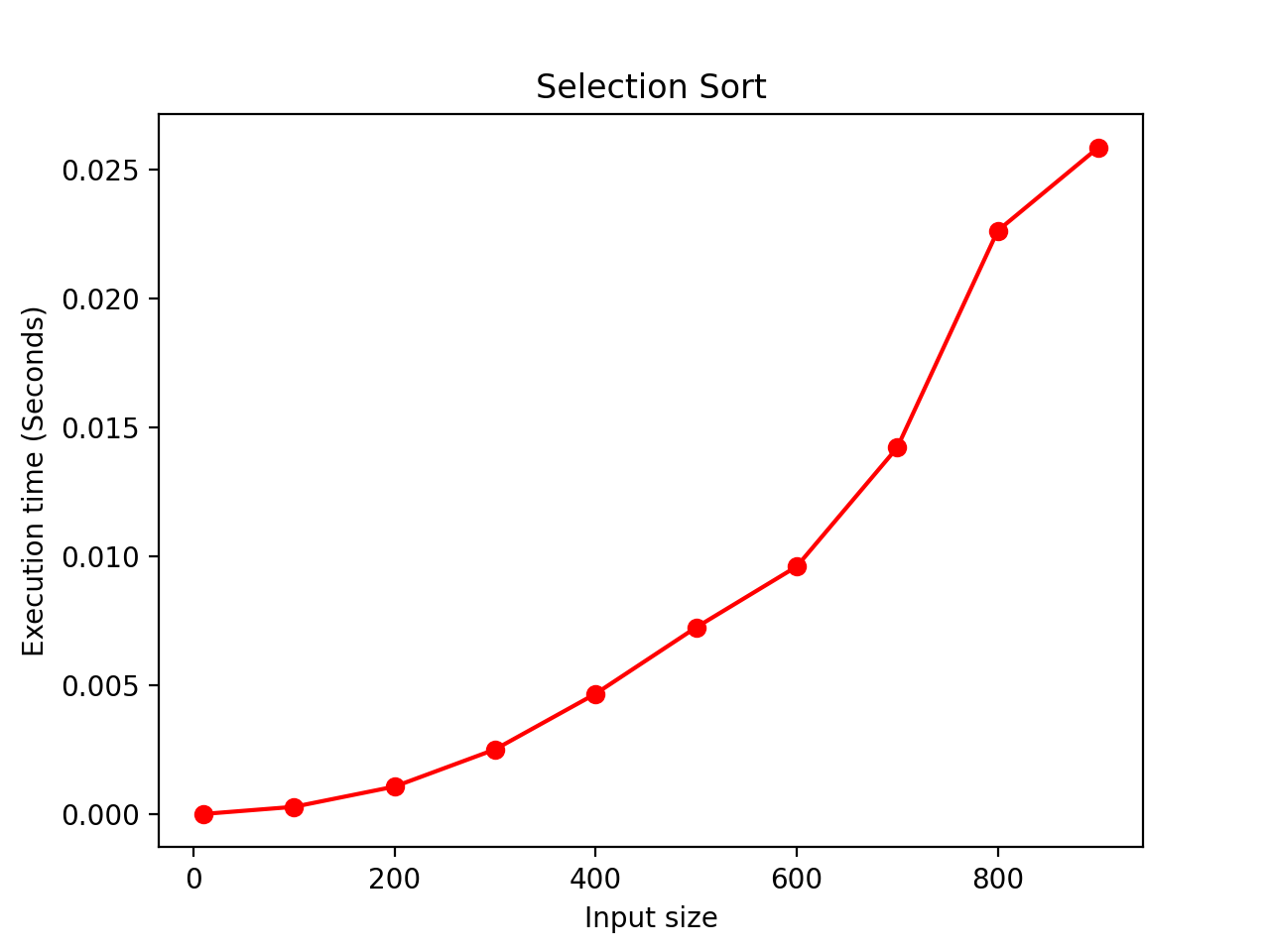
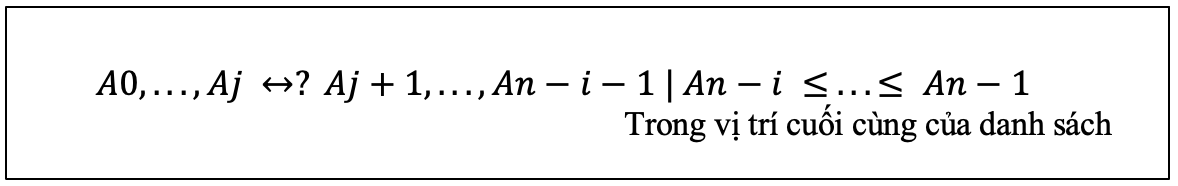


Figure 1:Biểu đồ thời gian thực của giải thuật Sắp xếp chọn

### Giải thuật Bubble Sort (Sắp xếp bong bóng):

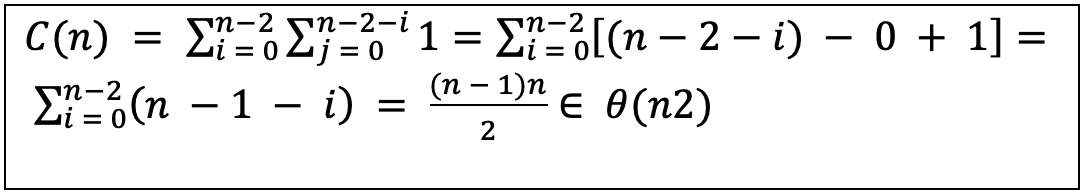
#### Ý tưởng giải thuật:

*  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:

#### Mã giả:

|  |
| --- |
| BubbleSort(A[**0.**.n − **1**])  # Sorts a given array by bubble sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Sorts a given array by bubble sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing orderimport time  def BubbleSort(*A*):  n = len(*A*)  for i in range(n-1):  for j in range(n-1-i):  if *A*[j+1] < *A*[j]:  *A*[j],*A*[j+1] = *A*[j+1],*A*[j]  A = [4,7,3,2,7,5,1]  BubbleSort(A)  print(A)  # [1, 2, 3, 4, 5, 7, 7] |

Code 2:Bubble Sort với python 3

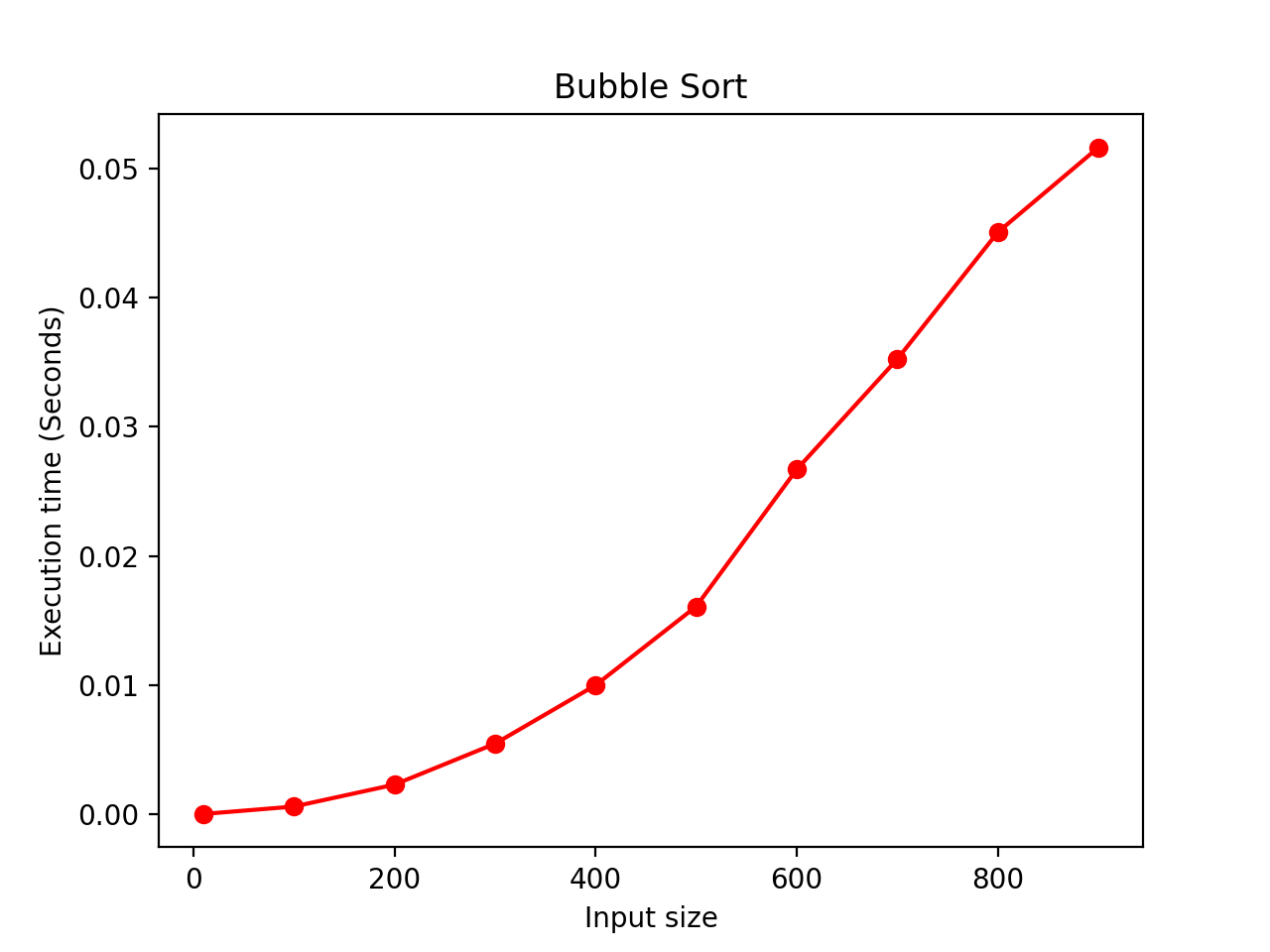


Figure 2:Biểu đồ thời gian thực của giải thuật Sắp xếp bóng bóng

### Giải thuật Sequential Search (Tìm kiếm tuần tự):

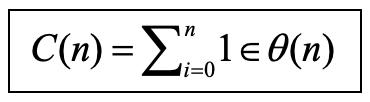
#### Ý tưởng giải thuật:

* 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:

|  |
| --- |
| SequentialSearch2(A[**0.**.n], K)  //Implements sequential search with a search key as a sentinel  //Input: An array A of n elements and a search key K  //Output: The index of the first element in A[0..n − 1] whose value  //is equal to K or −1 if no such element is found  A[n]← K  i ← **0**  **while** A[i]!= K do  i ← i + **1**  **if** i < n **return** i  **else** **return** −**1** |

#### Phân thích giải thuật:

* Tìm kiếm trong một danh sách như vậy có thể bị dừng ngay khi gặp phần tử lớn hơn hoặc bằng khóa tìm kiếm. Tìm kiếm tuần tự cung cấp một minh họa tuyệt vời về cách tiếp cận vét cạn, với điểm mạnh đặc trưng (tính đơn giản) và điểm yếu (hiệu quả kém hơn).

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Implements sequential search with a search key as a sentinel  # Input: An array A of n elements and a search key K  # Output: The index of the first element in A[0..n − 1] whose value is equal to K or −1 if no such element is found  def SequentialSearch(*A*,*k*):  i= 0  *A*.append(*k*)  n = len(*A*)-1  while *A*[i] != *k*:  i+=1  if i<n:  return i  else:  return -1  A = [4,7,3,2,7,5,1]  print(SequentialSearch(A,100))  # -1 |

Code 3:Sequential Search với python 3

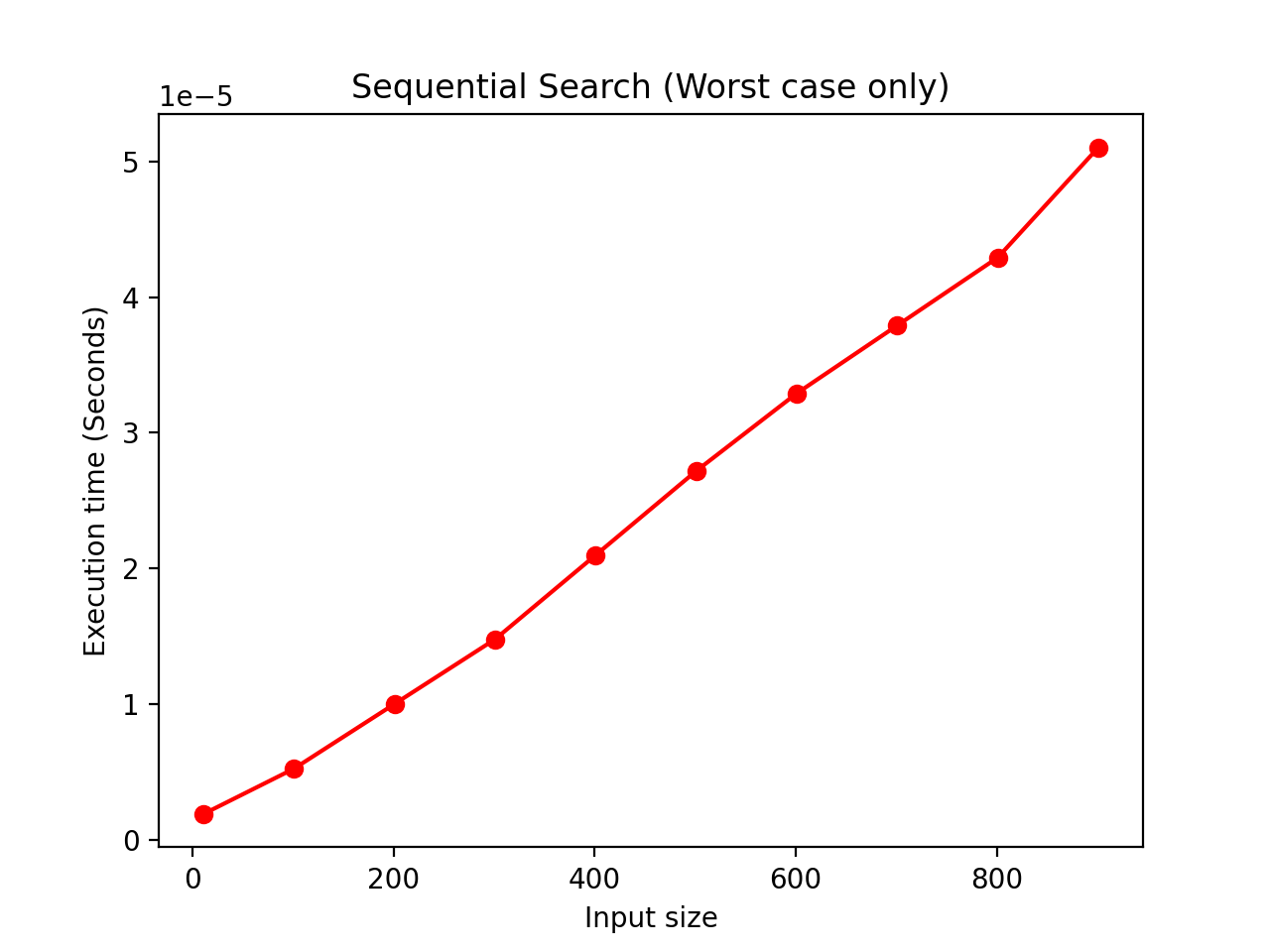


Figure 3:Biểu đồ thời gian thực của giải thuật Tìm kiếm tuần tự(chỉ trường hợp xấu nhất)

# CHƯƠNG II: THUẬT TOÁN DIVIDE AND CONQUER

## Giới thiệu về thuật toán Divide & Conquer:

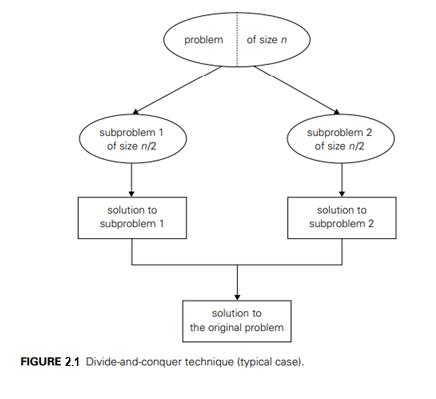
* Thuật toán phân chia và chinh phục (divide and conquer) là kỹ thuật thiết kế thuật toán chung được biết đến nhiều nhất. Mặc dù sự nổi tiếng của nó có thể liên quan đến cái tên hấp dẫn của nó, nhưng nó rất xứng đáng: một vài thuật toán khá hiệu quả là những triển khai cụ thể của chiến lược chung này.

## Ý thưởng của thuật toán:

* Các thuật toán phân chia và chinh phục hoạt động theo kế hoạch chung sau:

+ Một bài toán được chia thành nhiều bài toán con cùng loại, lý tưởng nhất là về kích thước bằng nhau.

+ Các bài toán con được giải quyết (thường là đệ quy, mặc dù đôi khi sử dụng một thuật toán khác, đặc biệt khi các bài toán con trở nên nhỏ đủ).

+ Nếu cần thiết, các giải pháp cho các bài toán con được kết hợp để có được một giải pháp đến vấn đề ban đầu.

## Các ví dụ cho thuật toán:

### Giải thuật Merge Sort (Sắp xếp hợp nhất):

#### Ý tưởng giải thuật:

* Sắp xếp hợp nhất (Merge Sort) là một ví dụ hoàn hảo cho ứng dụng thành công của kỉ thuật divide-and-conquer. Thuật toán sắp xếp một mãng A[0..n-1] cho trước bằng cách chia thành 2 mãng nhỏ A[0..n/2-1] và A[n/2 … n-1], sắp xếp 2 mãng bằng các đệ quy sau đó nối thành một mãng .Kết quả thu được một mãng được sấp xếp có thứ tự.

#### Mã giả:

|  |
| --- |
| Mergesort(A[**0.**.n − **1**])  //Sorts array A[**0.**.n − **1**] by recursive mergesort  //Input: An array A[**0.**.n − **1**] of orderable elements  //Output: Array A[**0.**.n − **1**] sorted **in** nondecreasing order  **if** n > **1**  copy A[**0.**.n/**2** − **1**] to B[**0.**.n/**2** − **1**]  copy A[n/**2**..n − **1**] to C[**0.**.n/**2** − **1**]  Mergesort(B[**0.**.n/**2** − **1**])  Mergesort(C[**0.**.n/**2** − **1**])  Merge(B, C, A)  Merge(B[**0.**.p − **1**], C[**0.**.q − **1**], A[**0.**.p + q − **1**])  //Merges two sorted arrays into one sorted array  //Input: Arrays B[**0.**.p − **1**] **and** C[**0.**.q − **1**] both sorted  //Output: Sorted array A[**0.**.p + q − **1**] of the elements of B **and** C  i ← **0**; j ← **0**; k ← **0**  **while** i < p **and** j < q do  **if** B[i]≤ C[j]  A[k]← B[i]; i ← i + **1**  **else** A[k]← C[j]; j ← j + **1**  k ← k + **1**  **if** i = p  copy C[j..q − **1**] to A[k..p + q − **1**]  **else** copy B[i..p − **1**] to A[k..p + q − **1**] |

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Sorts array A[0..n − 1] by recursive mergesort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order  def MergeSort(*A*):  n = len(*A*)  if(n>1):  B = *A*[:n//2].copy()  C = *A*[n//2:].copy()  MergeSort(B)  MergeSort(C)  Merge(B,C,*A*)  # Merges two sorted arrays into one sorted array  # Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted  # Output: Sorted array A[0..p + q − 1] of the elements of B and C  def Merge(*B*,*C*,*A*):  i=0  j=0  k=0  p = len(*B*)  q = len(*C*)  while i < p and j < q:  if *B*[i] <= *C*[j]:  *A*[k]=*B*[i]  i=i+1  else:  *A*[k]=*C*[j]  j=j+1  k=k+1  if i == p:  *A*[k:p+q]=*C*[j:q].copy()  else:  *A*[k:p+q]=*B*[i:p].copy()  A = [8, 3, 2, 9, 7, 1, 5, 4]  MergeSort(A)  print(A)  # [1, 2, 3, 4, 5, 7, 8, 9] |

Code 4:Merge Sort với python 3

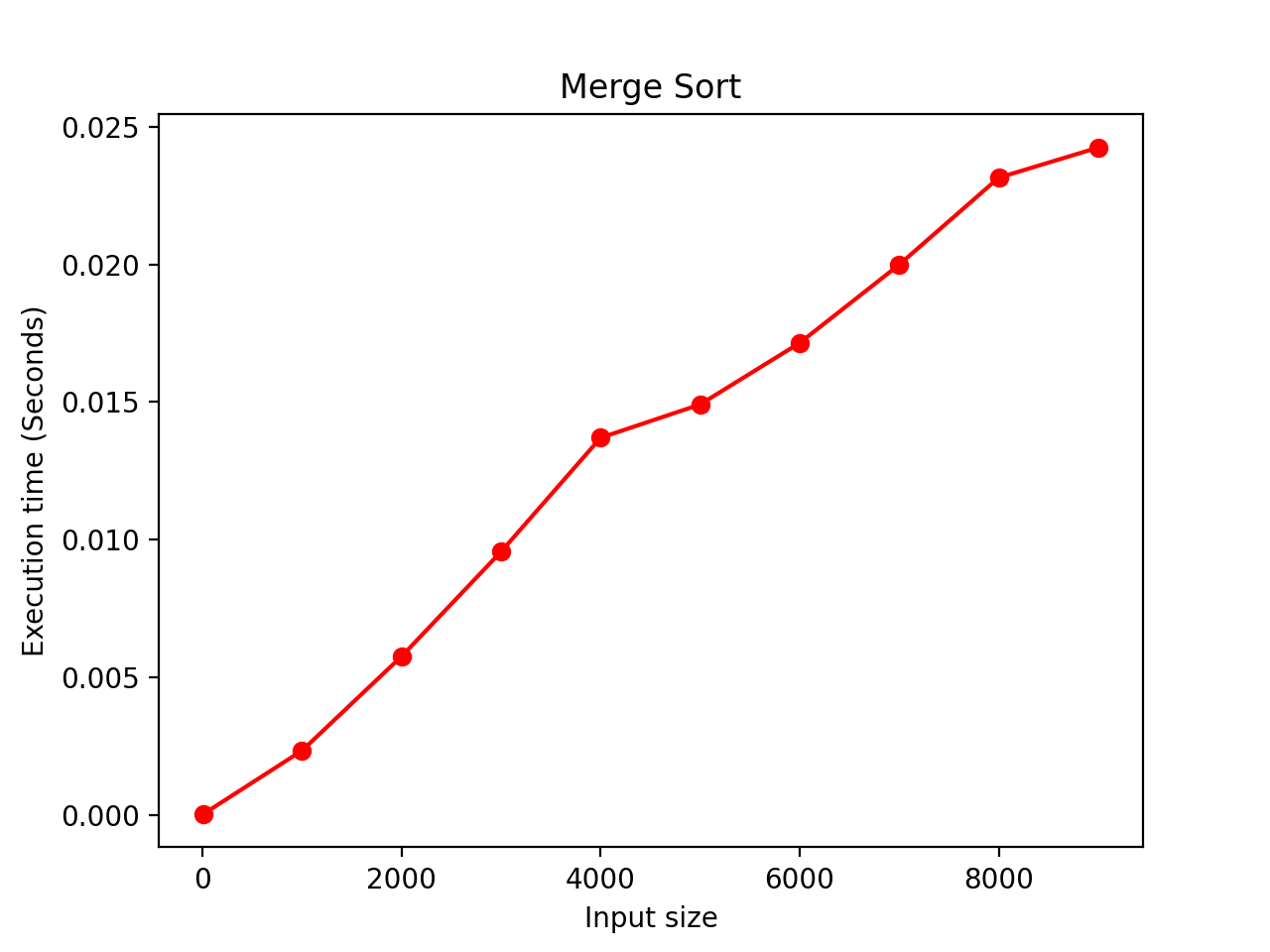


Figure 4:Biểu đồ thời gian thực của giải thuật Sắp xếp hợp nhất

### Giải thuật Quick Sort (Sắp xếp nhanh):

#### Ý tưởng giải thuật:

* Sắp xếp nhanh là một thuật toán sắp xếp quan trọng khác dựa trên cách tiếp cận chia theo tỷ lệ. Không giống như sắp xếp hợp nhất, phân chia các phần tử đầu vào của nó theo vị trí của chúng trong mảng, sắp xếp nhanh phân chia chúng theo giá trị của chúng. Phân hoạch là sự sắp xếp các phần tử của mảng sao cho tất cả các phần tử ở bên trái của một số phần tử A [s] nhỏ hơn hoặc bằng A [s] và tất cả các phần tử ở bên phải của A [s] đều lớn hơn hoặc bằng với nó. Sau khi đạt được phân vùng, A [s] sẽ ở vị trí cuối cùng của nó trong mảng đã sắp xếp và chúng ta có thể tiếp tục sắp xếp hai mảng con bên trái và bên phải của A [s] một cách độc lập (ví dụ: theo cùng một phương pháp).

#### Mã giả:

|  |
| --- |
| Quicksort(A[l..r])  //Sorts a subarray by quicksort  //Input: Subarray of array A[**0.**.n − **1**], defined by its left  //**and** right  // indices l **and** r  //Output: Subarray A[l..r] sorted **in** nondecreasing order  **if** l < r  s ←Partition(A[l..r]) //s **is** a split position  Quicksort(A[l..s − **1**])  Quicksort(A[s + **1.**.r])  HoarePartition(A[l..r])  //Partitions a subarray by Hoare’s algorithm, using the  //first element  // **as** a pivot  //Input: Subarray of array A[**0.**.n − **1**], defined by its left  //**and** right  // indices l **and** r (l < r)  //Output: Partition of A[l..r], **with** the split position  //returned **as**  // this function’s value  p ← A[l]  i ← l; j ← r + **1**  repeat  repeat i ← i + **1** until A[i]≥ p  repeat j ← j − **1** until A[j]≤ p  swap(A[i], A[j])  until i ≥ j  swap(A[i], A[j]) //undo last swap when i ≥ j  swap(A[l], A[j])  **return** j |

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Sorts a subarray by quicksort  # Input: Subarray of array A[0..n − 1], defined by its left and right  # indices l and r  # Output: Subarray A[l..r] sorted in nondecreasing order  def QuickSort(*A*,*l*,*r*):  if *l* < *r*:  s = Partition(*A*,*l*,*r*)  QuickSort(*A*,*l*,s-1)  QuickSort(*A*,s+1,*r*)  # Partitions a, using the last element as a pivot  # Input: Subarray of array A[0..n − 1], defined by its left and right  # indices l and r (l < r)  # Output: Partition of A[l..r], with the split position returned as  # this function’s value  def Partition(*A*,*l*,*r*):  p = *A*[*l*]  i = *l*  j = *r*+1  while True:  while True :  i = i+1  if (i >= len(*A*)-1):  break  else:  if (*A*[i] >=p):  break  while True:  j = j-1  if (j<= 0):  break  else:  if (*A*[j] <= p):  break  *A*[i],*A*[j] = *A*[j],*A*[i]  if(i>=j):  break  *A*[i],*A*[j] = *A*[j],*A*[i]  *A*[*l*],*A*[j] = *A*[j],*A*[*l*]  return j  A = [8, 3, 2, 9, 7, 1, 5, 4]  QuickSort(A,0,len(A)-1)  print(A)  # [1, 2, 3, 4, 5, 7, 8, 9] |

Code 5:Quick Sort với python 3

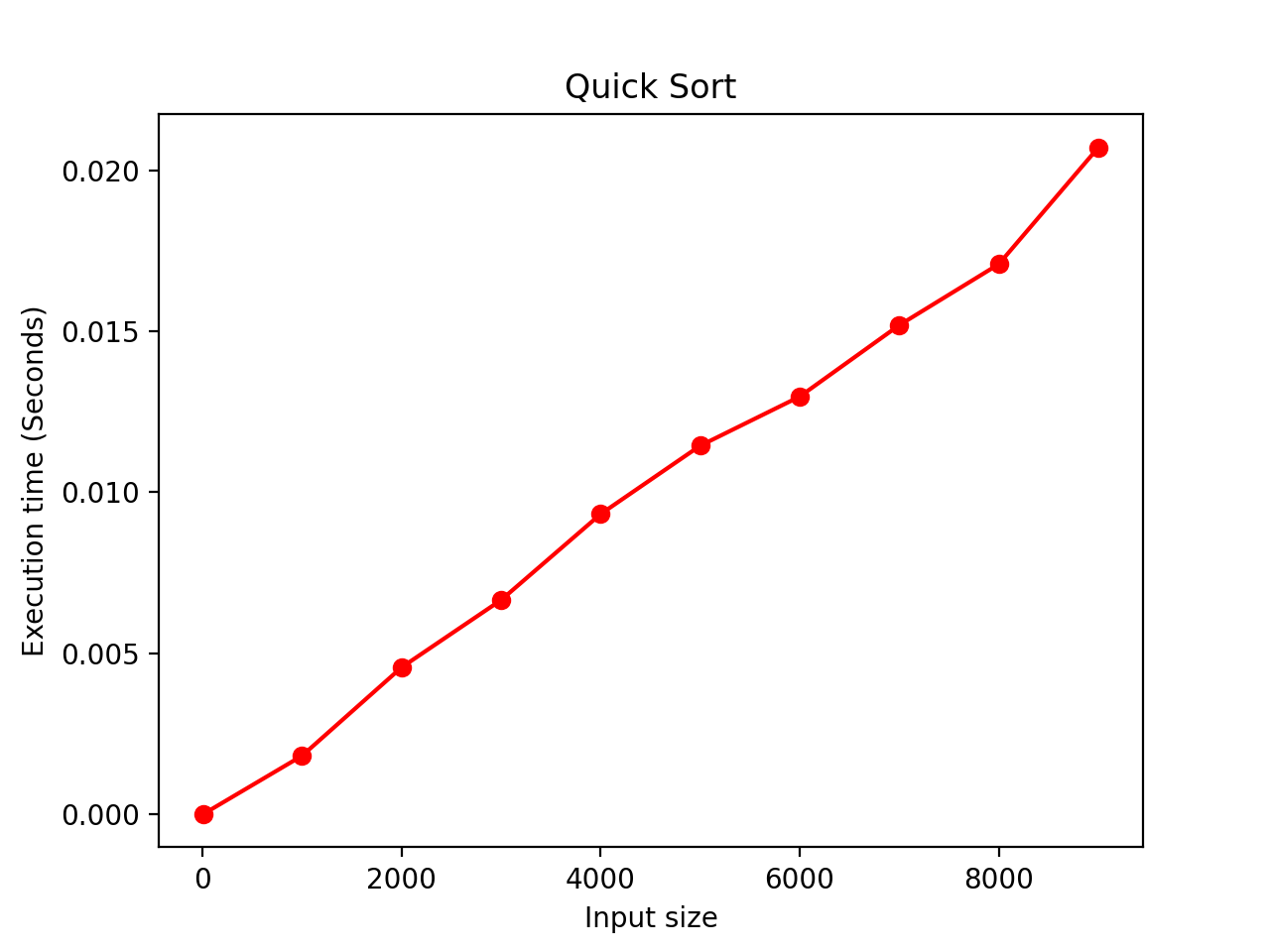
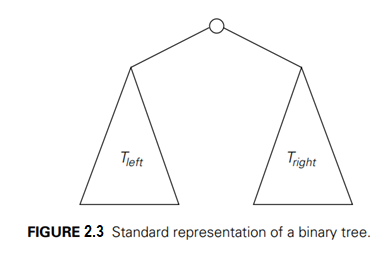


Figure 5:Biểu đồ thời gian thực của giải thuật Sắp xếp nhanh

### Giải thuật Binary Tree Traversals (Duyệt cây nhị phân):

#### Ý tưởng giải thuật:

* Cây nhị phân T được định nghĩa là một tập hợp hữu hạn các nút (nodes) trống hoặc bao gồn một gốc (root) và hai cây nhị phân rời rạc TL và TR được gọi là cây con trái và phải của gốc.



* Vì cây nhị phân chia thành 2 cấu trúc nhỏ hơn cùng loại, cây con bên trái và phải, nên chúng ta sẽ tính tổng chiều cao của cây bằng cách tính tổng chiều cao của cây con bên trái và phải sau đó cộng thêm 1.

#### Mã giả:

|  |
| --- |
| ﻿Height(T )  //Computes recursively the height of a binary tree  //Input: A binary tree T  //Output: The height of T  **if** T = ∅ **return** −1  **else** **return** max{Height(Tleft), Height(Tright)} + 1 |

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Computes recursively the height of a binary tree  # Input: A binary tree T  # Output: The height of T  def Height(*T*):  if *T* is None:  return -1  return max(Height(*T*.left), Height(*T*.right))+1  # A = [22, 20, 88, 84, 65, 10, 4, 75, 68, 6]  # print(Height(A))  # #3 |

Code 6: Binary Tree Traversals với python 3

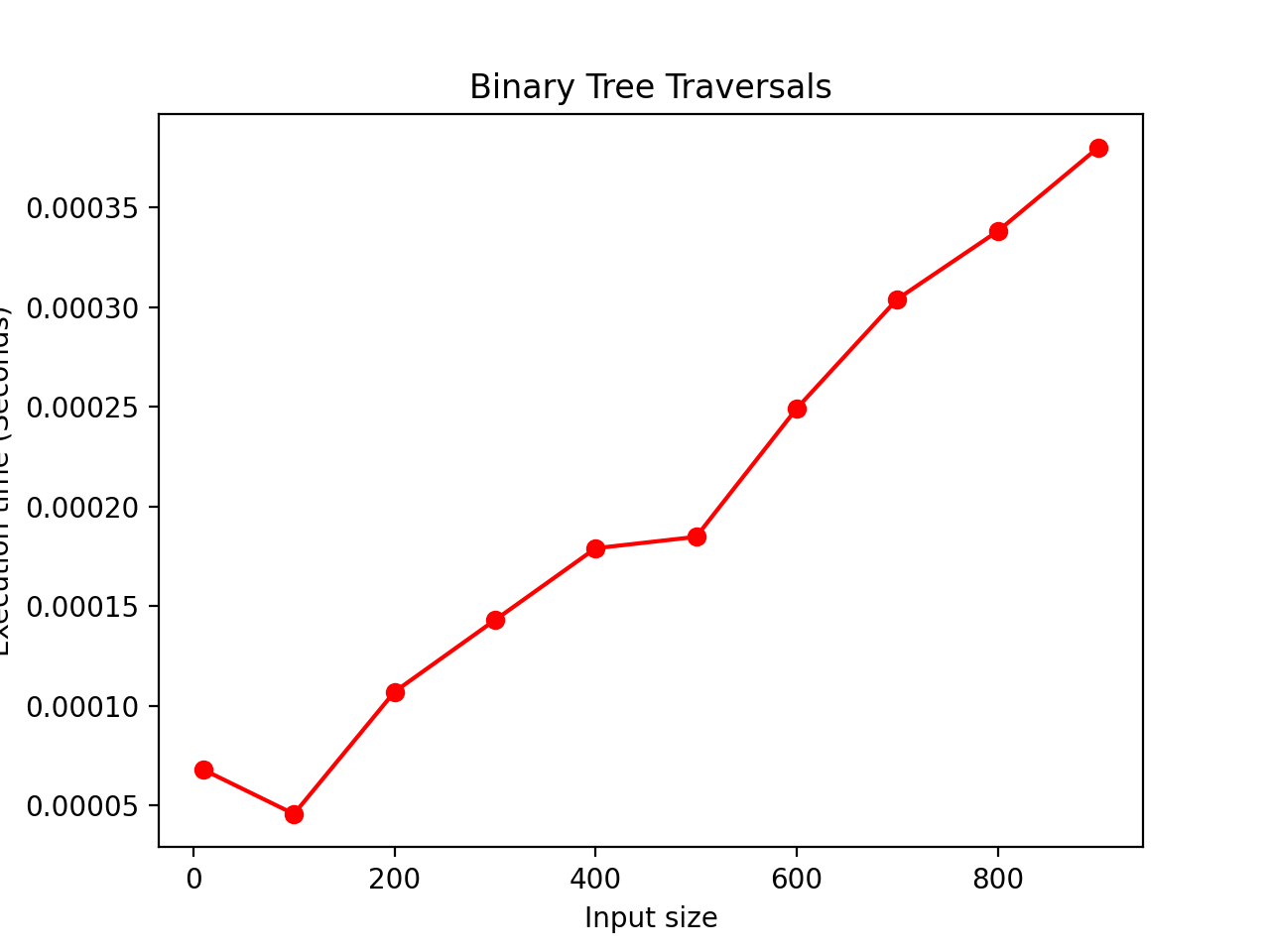


Figure 6:Biểu đồ thời gian thực của giải thuật Tìm chiều cao cây nhị phân

# CHƯƠNG III: THUẬT TOÁN GREEDY

## Giới thiệu về thuật toán Greedy:

* Thuật toán vét cạn (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.

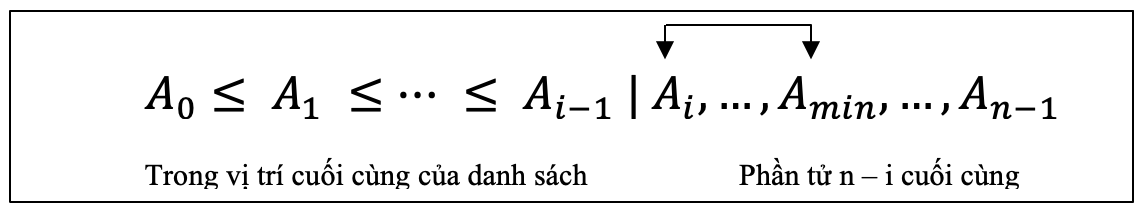
## Ý thưởng của thuật toán:

* “Đẩy tới” (force) bao hàm bởi định nghĩa của chiến lược là định nghĩa của máy tính chứ không phải là định nghĩa của trí tuệ.
* “Hãy làm đi” (just do it) có thể là một cách khác để mô tả qui tắc của các tiếp cận vét cạn. Và thông thường, chiến thuật vét cạn là một trong những cách đơn giản nhất được áp dụng.

## Các ví dụ cho thuật toán:

### Giải thuật Prim:

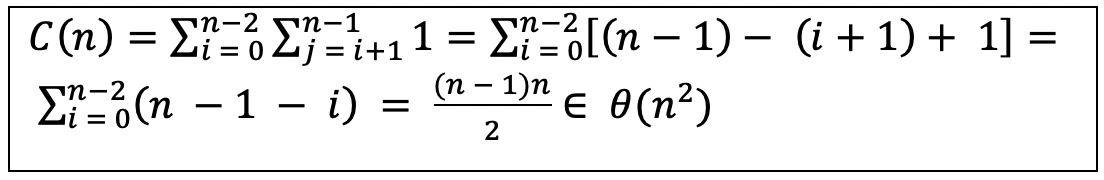
#### Ý tưởng giải thuật:

* 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:
* Sau khi quét n-1 lần, danh sách đã được sắp xếp.

#### Mã giả:

|  |
| --- |
| SelectionSort(A[**0.**.n−**1**])  # Sorts a given array by selection sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order  #import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Sorts a given array by selection sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing orderimport time  def SelectionSort(*A*):  n = len(*A*)  for i in range(n-1):  min = i  for j in range(i+1,n):  if *A*[j] < *A*[min]:  min = j  *A*[i],*A*[min] = *A*[min],*A*[i]  A = [4,7,3,2,7,5,1]  SelectionSort(A)  print(A)  # [1, 2, 3, 4, 5, 7, 7] |

Code 1:Selection Sort với python 3

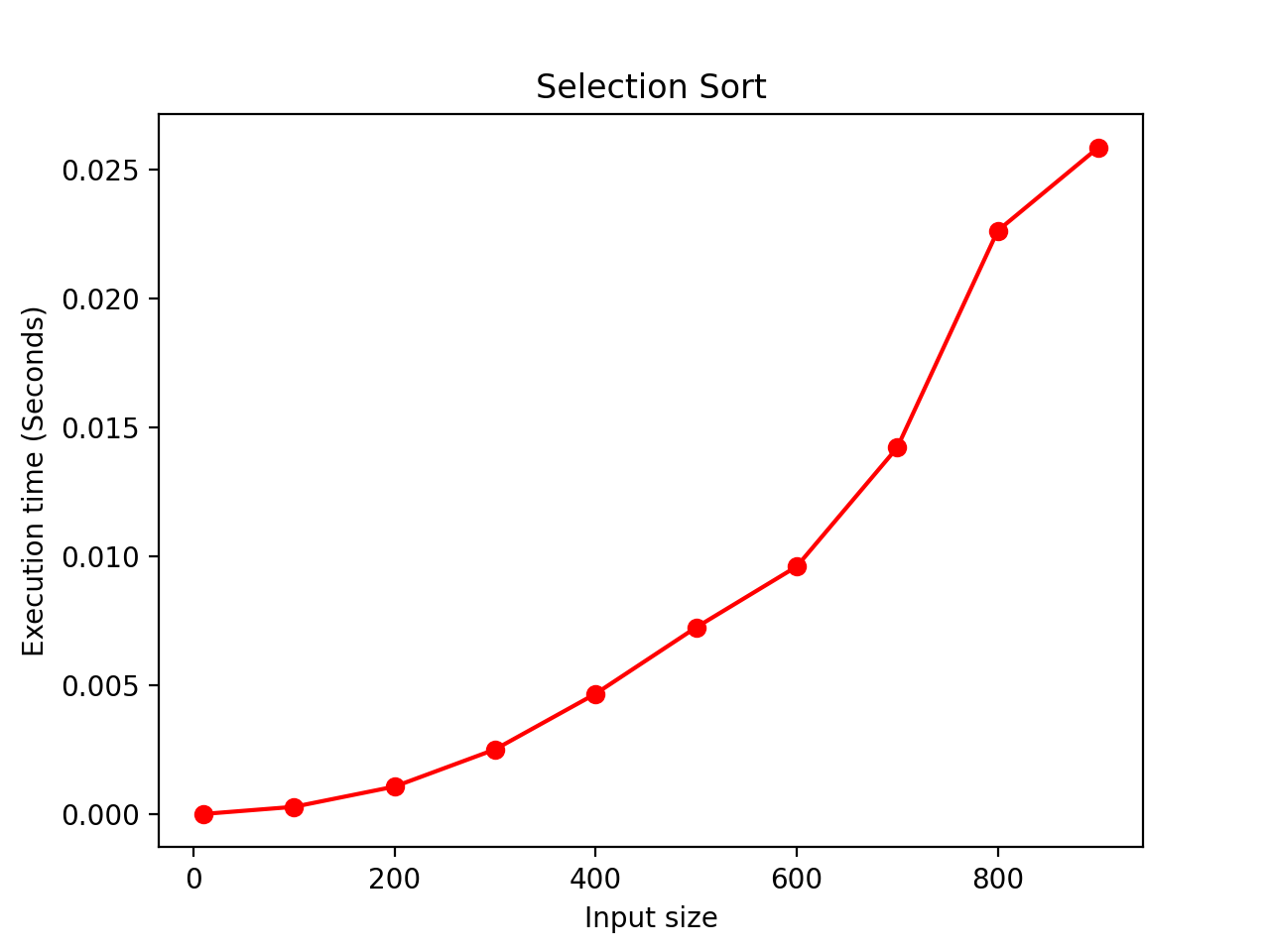
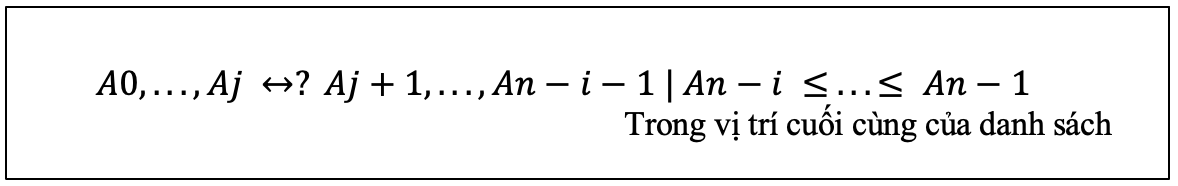


Figure 1:Biểu đồ thời gian thực của giải thuật Sắp xếp chọn

### Giải thuật Kruskal (Sắp xếp bong bóng):

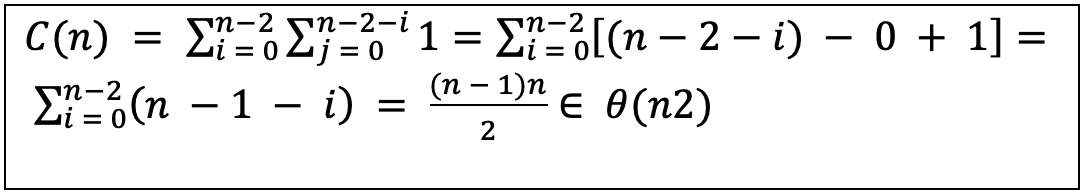
#### Ý tưởng giải thuật:

*  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:

#### Mã giả:

|  |
| --- |
| BubbleSort(A[**0.**.n − **1**])  # Sorts a given array by bubble sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Sorts a given array by bubble sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing orderimport time  def BubbleSort(*A*):  n = len(*A*)  for i in range(n-1):  for j in range(n-1-i):  if *A*[j+1] < *A*[j]:  *A*[j],*A*[j+1] = *A*[j+1],*A*[j]  A = [4,7,3,2,7,5,1]  BubbleSort(A)  print(A)  # [1, 2, 3, 4, 5, 7, 7] |

Code 2:Bubble Sort với python 3

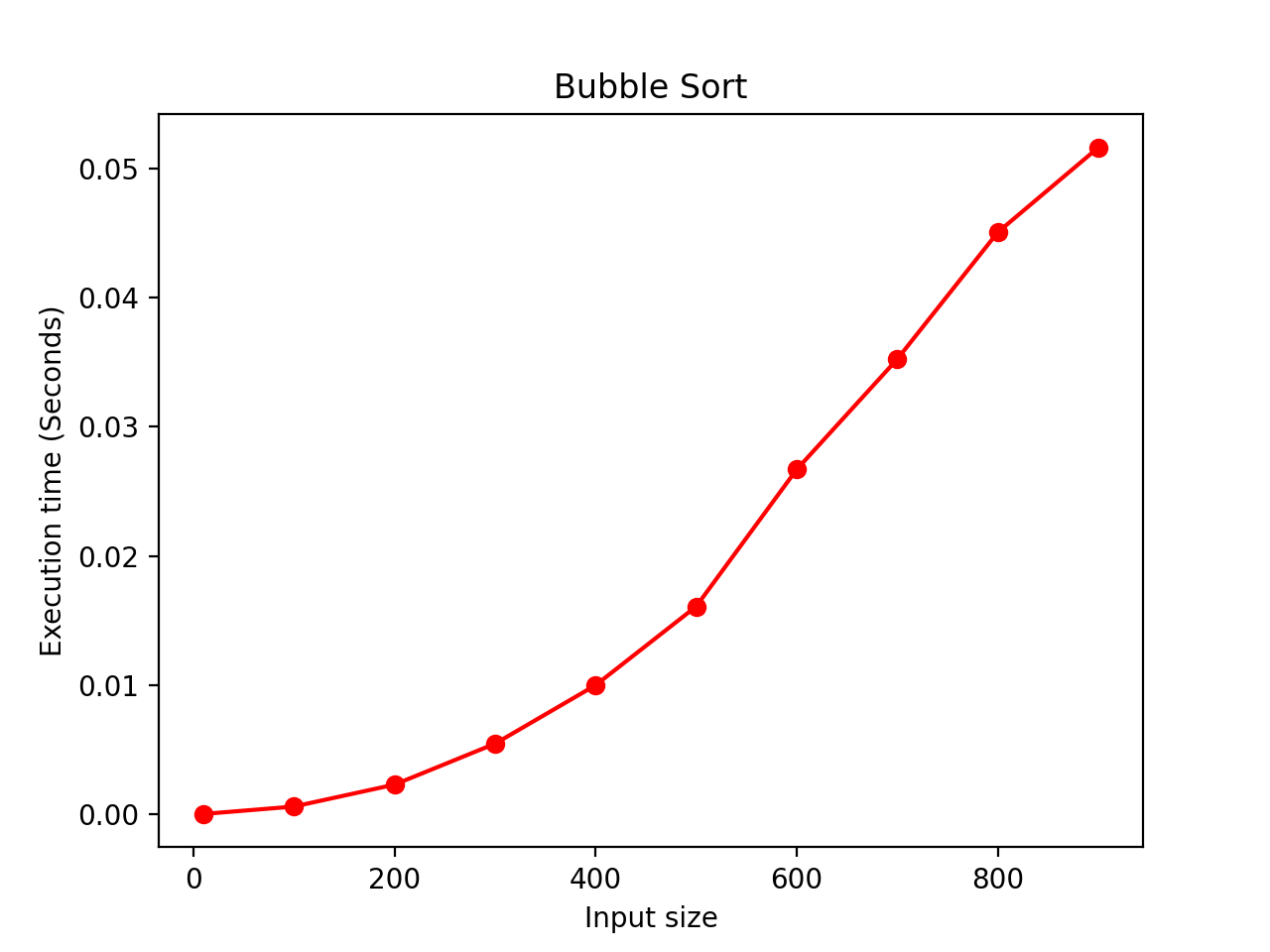


Figure 2:Biểu đồ thời gian thực của giải thuật Sắp xếp bóng bóng

### Giải thuật Dijkstra:

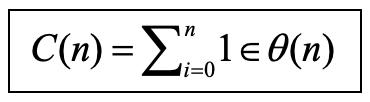
#### Ý tưởng giải thuật:

* 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:

|  |
| --- |
| SequentialSearch2(A[**0.**.n], K)  //Implements sequential search with a search key as a sentinel  //Input: An array A of n elements and a search key K  //Output: The index of the first element in A[0..n − 1] whose value  //is equal to K or −1 if no such element is found  A[n]← K  i ← **0**  **while** A[i]!= K do  i ← i + **1**  **if** i < n **return** i  **else** **return** −**1** |

#### Phân thích giải thuật:

* Tìm kiếm trong một danh sách như vậy có thể bị dừng ngay khi gặp phần tử lớn hơn hoặc bằng khóa tìm kiếm. Tìm kiếm tuần tự cung cấp một minh họa tuyệt vời về cách tiếp cận vét cạn, với điểm mạnh đặc trưng (tính đơn giản) và điểm yếu (hiệu quả kém hơn).

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Implements sequential search with a search key as a sentinel  # Input: An array A of n elements and a search key K  # Output: The index of the first element in A[0..n − 1] whose value is equal to K or −1 if no such element is found  def SequentialSearch(*A*,*k*):  i= 0  *A*.append(*k*)  n = len(*A*)-1  while *A*[i] != *k*:  i+=1  if i<n:  return i  else:  return -1  A = [4,7,3,2,7,5,1]  print(SequentialSearch(A,100))  # -1 |

Code 3:Sequential Search với python 3

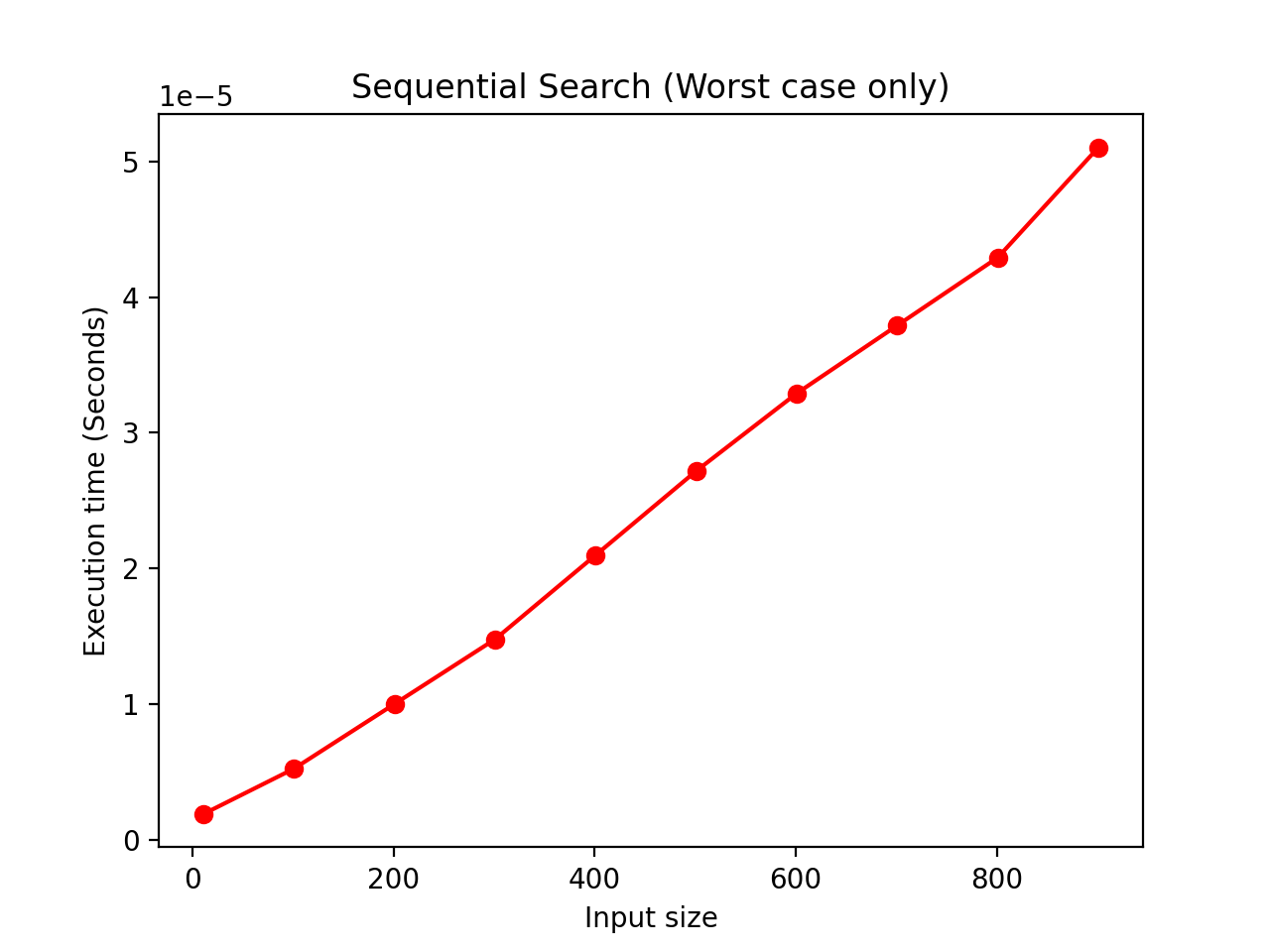


Figure 3:Biểu đồ thời gian thực của giải thuật Tìm kiếm tuần tự(chỉ trường hợp xấu nhất)

# CHƯƠNG IV: THUẬT TOÁN DYNAMIC PROGRAMMING

## Giới thiệu về thuật toán Dynamic Programming:

* Lập trình động (Dynamic Programming) là một kỹ thuật thiết kế thuật toán có lịch sử khá thú vị. Nó được phát minh bởi một nhà toán học Hoa Kỳ nổi tiếng, Richard Bellman, vào những năm 1950 như một phương pháp chung để tối ưu hóa các quy trình quyết định nhiều tầng.
* " Lập trình " trong tên của kỹ thuật này là viết tắt của "lập kế hoạch" và không đề cập đến lập trình máy tính. Sau khi chứng minh được giá trị của nó như một công cụ quan trọng của toán học ứng dụng, lập trình động cuối cùng đã được coi là một kỹ thuật thiết kế thuật toán tổng quát, ít nhất là trong giới khoa học máy tính, như một kỹ thuật thiết kế thuật toán chung không bị giới hạn trong các dạng bài toán tối ưu hóa đặc biệt.

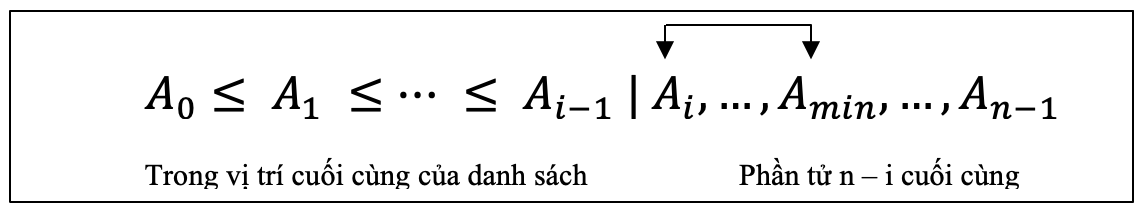
## Ý thưởng của thuật toán:

* Lập trình động là một kỹ thuật để giải quyết vấn đề với các bài toán con chồng chéo. Thông thường, các vấn đề con này phát sinh từ sự lặp lại liên quan đến giải pháp của một vấn đề nhất định với các giải pháp của các vấn đề con nhỏ hơn của nó. Thay vì giải các bài toán con chồng chéo lặp đi lặp lại, lập trình động đề xuất chỉ giải từng bài toán con nhỏ hơn một lần và ghi lại kết quả vào bảng từ đó có thể thu được lời giải cho bài toán ban đầu.

## Các ví dụ cho thuật toán:

### Giải thuật Selection Sort (Sắp xếp chọn):

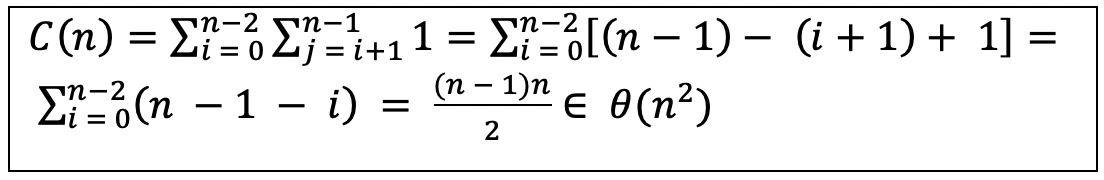
#### Ý tưởng giải thuật:

* 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:
* Sau khi quét n-1 lần, danh sách đã được sắp xếp.

#### Mã giả:

|  |
| --- |
| SelectionSort(A[**0.**.n−**1**])  # Sorts a given array by selection sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order  #import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Applies formula (8.3) bottom up to find the maximum amount of money  # that can be picked up from a coin row without picking two adjacent coins  # Input: Array C[1..n] of positive integers indicating the coin values  # Output: The maximum amount of money that can be picked up  def CoinRow(*C*):  *C*.insert(0, 0)  F = *C*  F[0] = 0  F[1] = *C*[1]  for i in range(2,len(*C*)):  F[i] = max(*C*[i]+F[i -2],F[i - 1])  return F[len(*C*)-1]  A = [5, 1, 2, 10, 6, 2]  print(CoinRow(A))  #10 |

Code 10:Coin Row với python 3

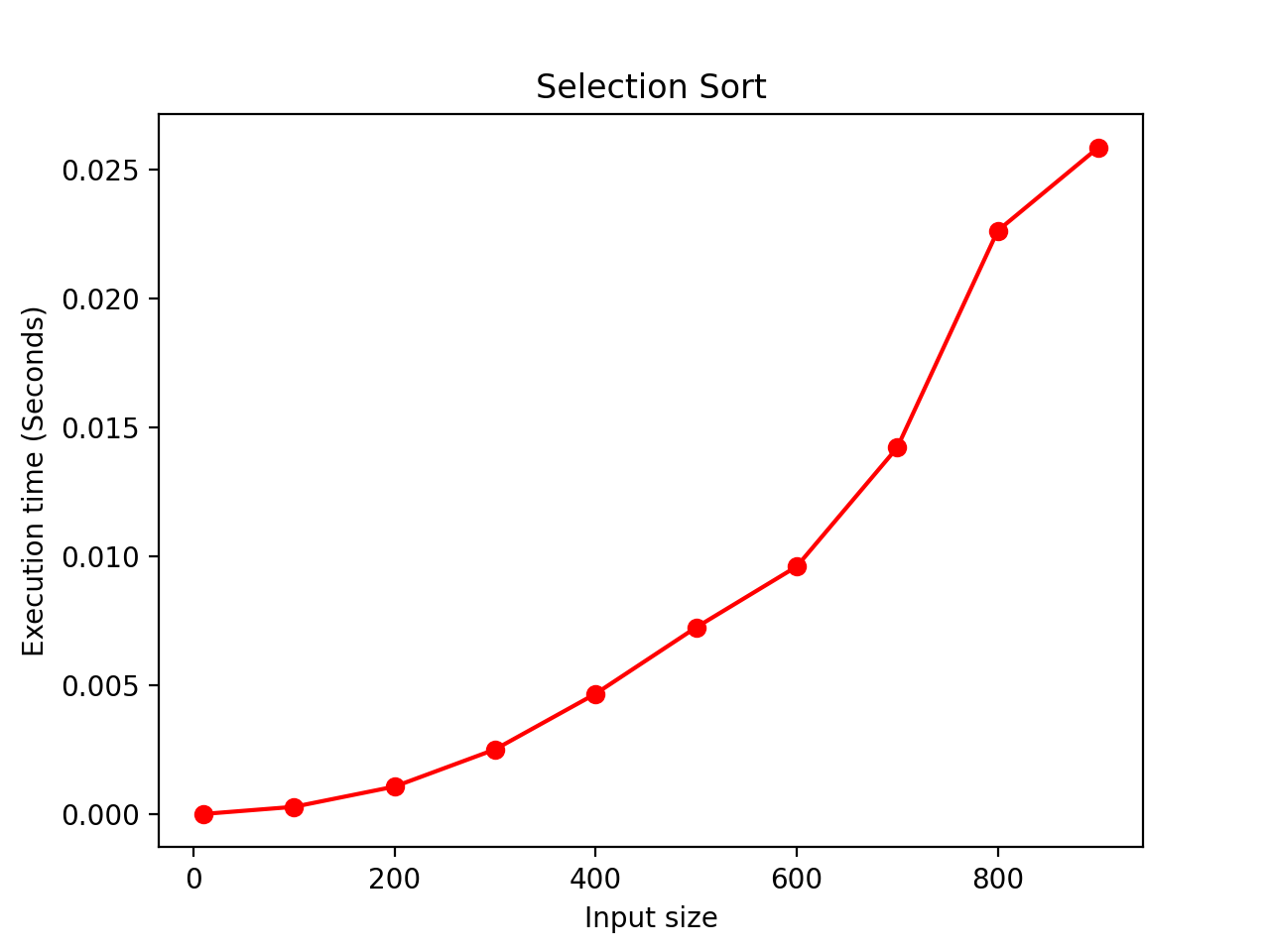
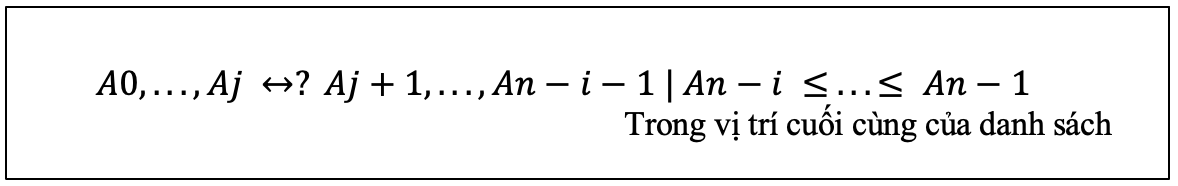


Figure 1:Biểu đồ thời gian thực của giải thuật Sắp xếp chọn

### Giải thuật Bubble Sort (Sắp xếp bong bóng):

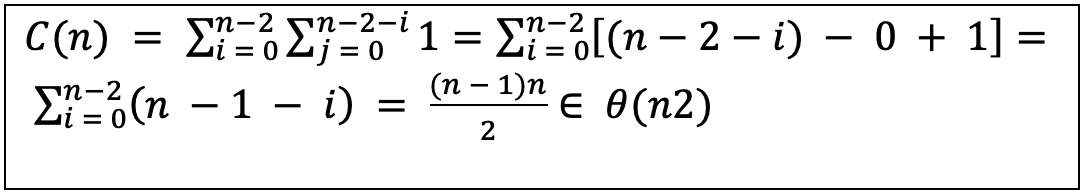
#### Ý tưởng giải thuật:

*  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:

#### Mã giả:

|  |
| --- |
| BubbleSort(A[**0.**.n − **1**])  # Sorts a given array by bubble sort  # Input: An array A[0..n − 1] of orderable elements  # Output: Array A[0..n − 1] sorted in nondecreasing order import time  **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 thích 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 giải thuật với python3:

|  |
| --- |
| # Applies dynamic programming to find the minimum number of coins  # of denominations d1 < d2 < . . . < dm where d1 = 1 that add up to a  # given amount n  # Input: Positive integer n and array D[1..m] of increasing positive  # integers indicating the coin denominations where D[1]= 1  # Output: The minimum number of coins that add up to n  def ChangeMaking(*D*,*n*):      m = len(*D*)-1      F = [0 for \_ in range(*n*+1)]      for i in range(1, *n*+1):          temp = 9999          j = 1          while j <= m and i>= *D*[j]:              temp = min(F[i-*D*[j]],temp)              j = j+1          F[i] = temp+1      return F[*n*]  A =[1,2,4]  print(ChangeMaking(A,20))  #5 |

Code 11:Coin Change với python 3

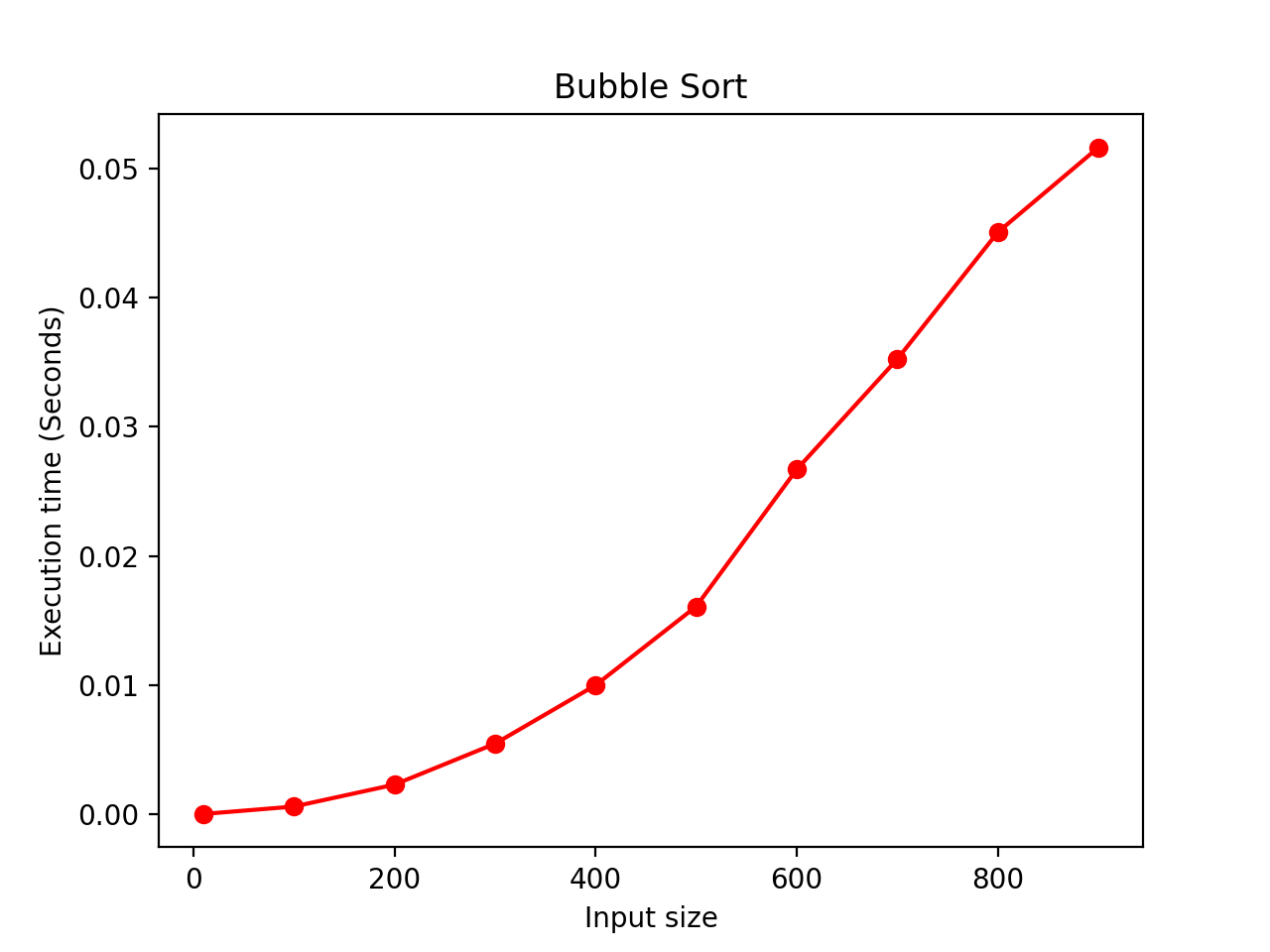


Figure 2:Biểu đồ thời gian thực của giải thuật Sắp xếp bóng bóng

### Giải thuật Sequential Search (Tìm kiếm tuần tự):

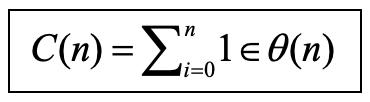
#### Ý tưởng giải thuật:

* 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:

|  |
| --- |
| SequentialSearch2(A[**0.**.n], K)  //Implements sequential search with a search key as a sentinel  //Input: An array A of n elements and a search key K  //Output: The index of the first element in A[0..n − 1] whose value  //is equal to K or −1 if no such element is found  A[n]← K  i ← **0**  **while** A[i]!= K do  i ← i + **1**  **if** i < n **return** i  **else** **return** −**1** |

#### Phân thích giải thuật:

* Tìm kiếm trong một danh sách như vậy có thể bị dừng ngay khi gặp phần tử lớn hơn hoặc bằng khóa tìm kiếm. Tìm kiếm tuần tự cung cấp một minh họa tuyệt vời về cách tiếp cận vét cạn, với điểm mạnh đặc trưng (tính đơn giản) và điểm yếu (hiệu quả kém hơn).

#### Triển khai giải thuật với python3:

|  |
| --- |
| # Implements sequential search with a search key as a sentinel  # Input: An array A of n elements and a search key K  # Output: The index of the first element in A[0..n − 1] whose value is equal to K or −1 if no such element is found  def SequentialSearch(*A*,*k*):  i= 0  *A*.append(*k*)  n = len(*A*)-1  while *A*[i] != *k*:  i+=1  if i<n:  return i  else:  return -1  A = [4,7,3,2,7,5,1]  print(SequentialSearch(A,100))  # -1 |

Code 3:Sequential Search với python 3

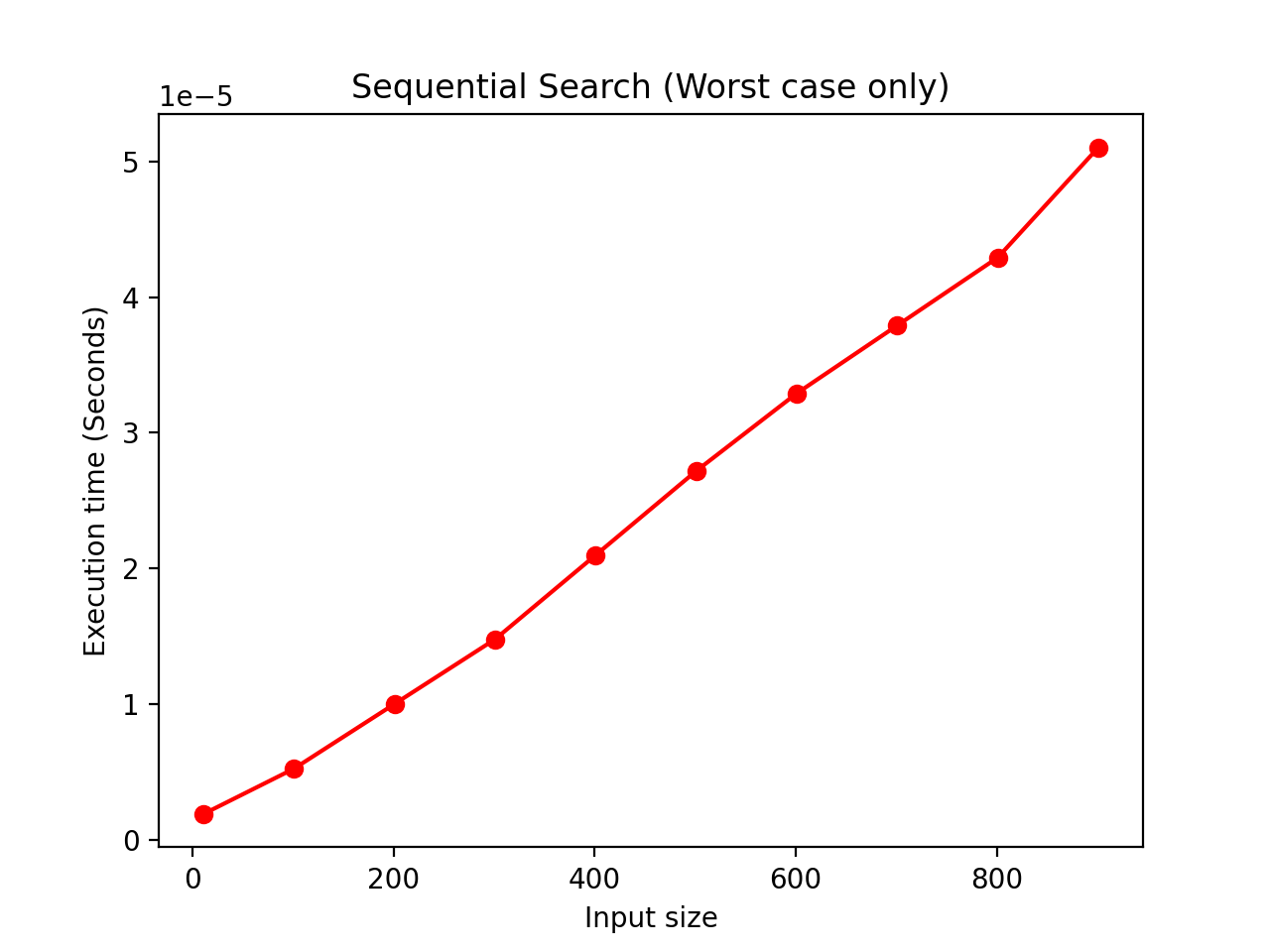


Figure 3:Biểu đồ thời gian thực của giải thuật Tìm kiếm tuần tự(chỉ trường hợp xấu nhất)