# Brute force

**Định nghĩa phương pháp**

* Brute force là một cách tiếp cận các cách giải quyết nó vấn đề một cách đơn giản nhất, thường là trực tiếp dựa trên định nghĩa của các khái niệm được nhắc tới.

**Ý tưởng của phương pháp**

* "Force" được ngụ ý là một chiến lược của máy tính chứ không phải trí tuệ con người.
* "Cứ làm đi!" sẽ là một cách nói khác để miêu tả Brute force là gì, miễn giải ra kết quả là ổn và chúng không nhất thiết phải là cách làm tối ưu nhất.

**Các ví dụ giải thuật cho phương pháp giải Brute force**

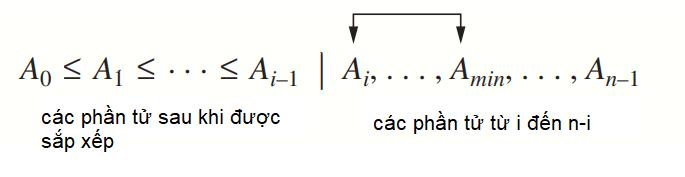
* Dựa theo cuốn sách tham khảo, phần này chúng ta sẽ nói xem xét sự áp dụng của chiến lược Brute force thông qua các giải thuật sắp xếp (sorting)
* Cho một dãy n phần tử có thứ tự (ví dụ như bảng chữ cái) và bây giờ hãy sắp xếp nó theo hướng giảm dần độ ưu tiên của từng chữ cái
* Theo cuốn sách đề cập, ở mục 1.3 đã để cập đến 1 số giải thuật có liên quan nhưng để nắm rõ nhất vấn đề thì hãy xem như chưa từng đọc mục này đi rồi đặt vấn đề là: Phương pháp đơn giản nhất sẽ là gì để giải quyết vấn này? Một số người có kinh nghiệm, họ sẽ không đồng ý với lối nghĩ này nhưng nó sẽ thật hợp lý khi chúng ta phải giải quyết nhanh chóng 1 công việc nào đó bằng cách đưa ra phương án nhanh nhất có thể.

## Selection sort (Sắp xếp chọn)

**Ý tưởng thuật giải:**

Chúng ta bắt đầu thuật toán sắp xếp chọn bằng cách quét toàn bộ mảng cho trước để tìm phần tử nhỏ nhất của nó và đổi nó với phần tử đầu tiên lúc này chúng ta đã đặt phần tử nhỏ nhất này vào vị trí cuối cùng của nó trong mảng đã được sắp xếp

Sau đó, chúng ta tiếp tục duyệt mảng với các phần tử còn lại, bắt đầu với phần tử thứ hai, để tìm phần tử nhỏ nhất trong số n - 1 phần tử còn lại với n là độ dài mảng cho trước và trao đổi nó với phần tử thứ hai, đặt phần tử nhỏ thứ hai vào vị trí cuối cùng của nó. Nói chung, phần tử thứ i-th mỗi khi t duyệt trong mảng, thuật toán sẽ tìm phần tử nhỏ nhất trong khoảng n-i phần tử và đổi nó với phần tử thứ i của mảng



**Mã giả - Pseudo code:**

ALGORITHM 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

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 của giải thuật thực ra là số phần tử n; phép toán cơ bản đắt giá nhất là so sánh khóa A [j] <A [min] được đánh dấu ở (\*).

Trường hợp tốt nhất là khi mảng có 1 phần tử hoặc không có phần tử khi giải thuật không cần thực hiện bất kì phép lặp nào

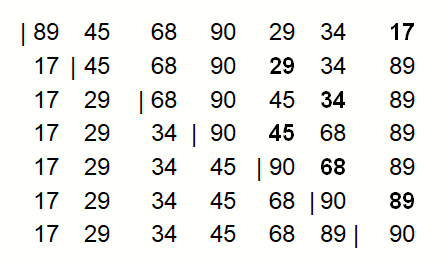
Trường hợp xấu nhất cũng như trường hợp trung bình là khi giải thuật phải duyệt hết mảng cho trước.

Còn số lần nó được thực thi chỉ phụ thuộc vào kích thước mảng và được cung cấp bởi tổng sau:



**Mô tả bước chạy:**

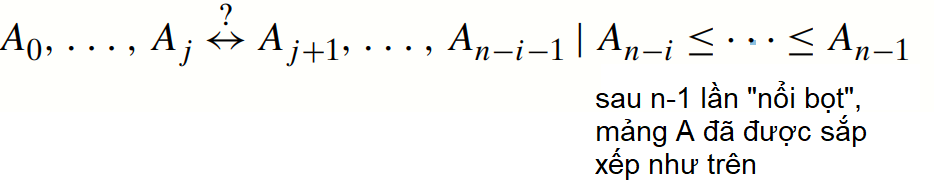
* Cho mảng A = {89, 45, 68, 90, 29, 34, 17 } thì các bước chạy của sắp xếp chọn sẽ như sau:

****

## Bubble sort (Sắp xếp nổi bọt biển)

**Ý tưởng thuật giải:**

* Sắp xếp nổi bọt là so sánh các phần tử liền kề của phần tử trong mảng các phần tử và trao đổi chúng nếu chúng không theo thứ tự.
* Bằng cách thực hiện lặp đi lặp lại, cuối cùng chúng tôi sẽ “nổi bọt” phần tử lớn nhất lên vị trí cuối cùng của nó trong mảng. Tiếp đến là lần nổi bong bóng cho phần tử lớn thứ hai, và cứ tiếp tục như vậy cho đến khi sau n - 1 lần "nổi bọt" như vậy thì mảng được sắp xếp
* Các lượt đi của phần tử i trong mảng phần tử A với  sẽ được biểu hiện như sau:



**Mã giả - Pseudo code:**

ALGORITHM *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 non- decreasing order

**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:**

Trường hợp tốt nhất là khi mảng có 1 phần tử hoặc không có phần tử khi giải thuật không cần thực hiện bất kì phép lặp nào

Trường hợp xấu nhất cũng như trường hợp trung bình là khi giải thuật phải duyệt hết mảng cho trước.

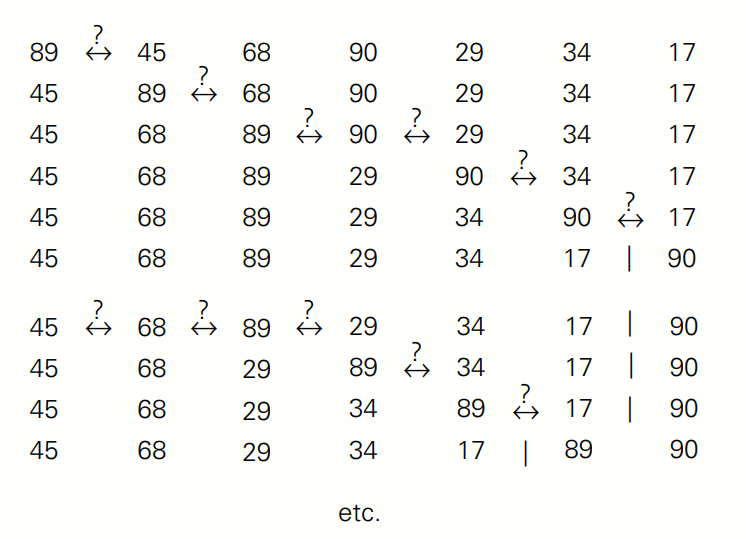
Xác đinh được phép tính cơ bản đắt giá nằm ở phép so sánh của dòng thứ 3 và vòng lặp for thứ 2 sẽ là phép lặp nhiều nhất

Còn số lần nó được thực thi chỉ phụ thuộc vào kích thước mảng và được cung cấp bởi tổng sau:



**Mô tả bước chạy:**

* Cho mảng A = {89, 45, 68, 90, 29, 34, 17 } thì các bước chạy của sắp xếp chọn sẽ như sau:

****

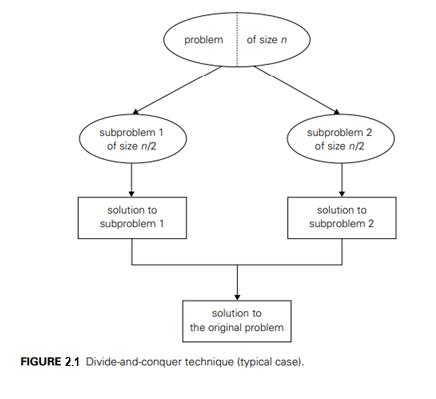
1. **Giới thiệu**

Kỹ thuật Divede-and-conquer thường được biết đến là một kỹ thuật thiết kế thuật toán chung vì tính hiệu quả của thuật toán được thực thi một cách cụ thể có chiến lược chung.

1. **Cách hoạt động**

Kỹ thuật Divede-and-conquer hoạt động dựa theo chiến lược chung bao gồm:

1. Vấn đề được chia nhỏ thành các vấn để con cùng loại, lý tưởng nhất là có cùng 1 kích thước
2. Giải quyết các vấn để nhỏ ( thường là sử dụng đệ quy, trong một vài trường hợp vấn đề con đủ nhỏ thì sẽ sử dụng thuật toán khác )
3. Nếu cấn thiết, các giải pháp của các vấn để nhỏ sẽ được kết hợp để có được một giải pháp chung cho vấn đề ban đầu.

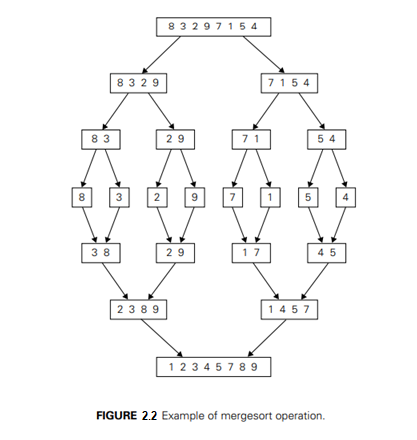


**3. Ví dụ**

**1) Mergesort**

**a) Ý tưởng**

Mergesort 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.



**Psuedocode:**

**ALGORITHM** *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]

The operation of the algorithm on the list 8*,* 3*,* 2*,* 9*,* 7*,* 1*,* 5*,* 4 is illustrated in

**Thực thi**

**Demo**

**2)Quicksort**

**Ý tưởng**

Quicksort là thuật toán quan trọng sắp xếp khác dựa trên divide-and-conquer. Không giống mergesort, thuật toán chia các phần tử đầu vào theo vị trí trong mãng, quicksort chia theo giá trị.

**Psuedocode**

**ALGORITHM** 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])

def Partition(A[l…r]):

pivot = A[l]

h= l

for k ← l+1 … r, do:

if A[l…r] < pivot:

h+=1

swap( A[k], A[h] )

swap(A[h], A[i] )

return h

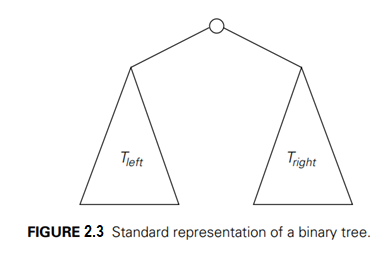
**Thực thi**

**Demo**

**3. Binary Tree Traversals and Related Properties**

Binary tree T được định nghĩa là một tập hữu hạn của nodes, tập đó có thể rỗng hoặc chứa rễ và hai binary tree riêng biệt TL và TR cho mỗi bên trái và phải. Chiều cao binary tree là độ dài lớn nhất từ rễ đến lá.Các vấn đề về binary tree có thể được giải quyết bằng các áp dụng kỹ thuật divide-and-conquer.

Ví dụ cụ thể là tính chiều cao height của cây. Height có thể được tính là độ dài lớn nhất của rễ bên trái và phải cây con cộng. Tuy nhiên để thuận tiện ta có thể gán chiều dài cho cây rỗng là -1.



**Psuedocode**

**ALGORITHM** 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(T\_left), Height(T\_right)} + 1

# Dynamic Programming

* 1. **giới thiệu**

Dynamic programming là một kỹ thuật thiết kế thuật toán có lịch sử thú vị. Được phát minh bởi nhà toán học Richard Bellman ở U.S vào năm 1950. Kỹ thuật nào là một giải pháp chung cho việc tối ưu hóa quá trình ra quyết định.

**2.** **Ý TƯỞNG**

Dynamic programming là một kỹ thuật xử lý problem bằng cách nối chồng (overlapping) các subproblem. Thông thường là các subproblem tăng lên từ việc lập lại các giải pháp của một vấn đề cho đến giải pháp của một vấn đề nhỏ hơn .

Kỹ thuật này đề xuất xử lý mỗi smaller subproblem chỉ một lần và ghi kết quả vào bảng. Bảng bao gồm từ giải pháp cho đến vấn để ban đầu có thể đạt được.

**3.Examples**

Giả sử có 1 hàng gồm n đồng xu, mỗi đồng xu có giá trị là các số nguyên dương.

**1 ) problem 1: Coin-row**

Chọn giá trị lớn nhất của n đồng xu với điều kiện kết quả chọn không có 2 đồng xu liền kề trong hàng

**a) Idea**

#Input: n đồng xu với mỗi đồng có giá trị nguyên dương C1, C2 … Cn

#Output: Giá trị lớn nhất sao kết quả chọn không có 2 đồng xu liền kề trong hàng

Cho F(n) là số lượng tối đa đồng xu ( nhiều nhất là n ). Để thu được vòng lập cho F(n), Chúng ta phân vùng tất cả các đồng xu được chọn thành 2 nhóm:

* Nhóm 1: nhóm đồng xu bao gồm đồng xu cuối cùng
* Nhóm 2: nhóm đồng xu không bao gồm đồng xu cuối cùng

Số tiền lớn nhất ta có thể thu được từ nhóm 1 : **cn + F( n-2 ) .**Giá trị của đồng xu nth cộng với giá trị lớn nhất ta có thể chọn từ [ 1 … n-2 ].

Số tiền lớn nhất ta có thể thu được từ nhóm 2: **F(n-1) .**

Từ đối tượng recurrence và điều kiện ta có:

**F(n) = max{cn + F (n − 2), F (n − 1)},** với n > 1,

**F (0) = 0,**

**F(1) = c1.**

**Psuedocode:**

**ALGORITHM** CoinRow(C[1..n])

//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

F[0]← 0; F[1]← C[1]

for i ← 2 to n do :

F[i]← max(C[i] + F[i − 2], F[i − 1])

return F[n]

**b) problem 2: Change-making**

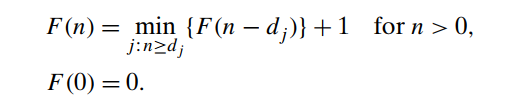
Chọn ra số lượng đồng xu tối thiếu có thể tổng bằng **n**

**Idea**

Cho F(N) là số lượng tối thiếu đồng xu có tổng giá trị bằng n.

n : là số tiền cần phải đạt được.

Số tiền n có thể đạt được bằng các cộng từng đồng có giá trị dj cho đến ( n - dj ) với j [1, 2, …, m ] và n >= dj



**ALGORITHM** ChangeMaking(D[1..m], n)

//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

F[0]← 0

for i ← 1 to n do

temp ← ∞; j ← 1

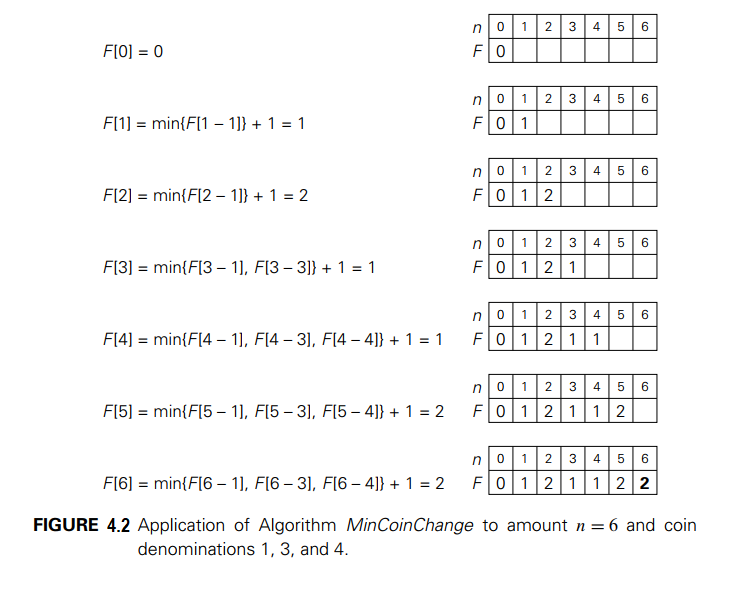
while j ≤ m and i ≥ D[j ] do

temp ← min(F[i − D[j ]], temp)

j ← j + 1

F[i]← temp + 1

return F[n]



**c) problem 3: Coin-Collecting problem**

Một bảng *n* × *m,* mỗi ô chỉ có một đồng xu hoặc không có đồng xu nào. Một robot được đặt ở vị trí góc trái bên trên có nhiệm vụ thu thập đồng xu nhiều nhất có thể từ vị trí bắt đầu đến gốc phải bên dưới. Mỗi bước robot sẽ di chuyển từ vị trí hiện tới ô bên phải HOẶC xuống ô bên dưới.

Thiết kế một thuật toán tìm số lượng đồng xu nhiều nhất mà robot có thể tìm.

**Idea**

F( i, j) : số đồng xu lớn nhất robot có thể thu thập và mang tới ô ở dòng ith , cột jth . ….