Mục lục Báo cáo thuật toán 2

[**I.** **Mô hình thuật toán tham lam (Greedy Algorithrm)** 2](#_Toc133991040)

[**1.** **Ý nghĩa và cách hoạt động của thuật toán.** 2](#_Toc133991041)

[**2.** **Ví dụ** 3](#_Toc133991042)

[**3.** **Ứng dụng của thuật toán** 4](#_Toc133991043)

[**4.** **Đánh giá thuật toán** 4](#_Toc133991044)

[**5.** **CODE PTIT** 4](#_Toc133991045)

[**II.** **Mô hình thuật toán chia và trị (Devide and conquer Algorithm)** 5](#_Toc133991046)

[**1.** **Ý nghĩa và cách hoạt động của thuật toán.** 5](#_Toc133991047)

[**2.** **Ví dụ** 6](#_Toc133991048)

[**3.** **Ứng dụng của thuật toán** 9](#_Toc133991049)

[**4.** **Đánh giá thuật toán** 9](#_Toc133991050)

[**5.** **CODE PTIT** 9](#_Toc133991051)

[**III.** **Mô hình thuật toán qui hoạch động (Dynamic Programming Algorithrm)** 10](#_Toc133991052)

[**1.** **Ý nghĩa và cách hoạt động của thuật toán** 10](#_Toc133991053)

[**2.** **Ví dụ** 12](#_Toc133991054)

[**3.** **Ứng dụng của thuật toán** 16](#_Toc133991055)

[**4.** **Đánh giá thuật toán** 16](#_Toc133991056)

[**5.** **CODE PTIT** 16](#_Toc133991057)

[**IV.** **So sánh 3 thuật toán Greedy, Divide And Conquer, Dynamic Programming Algorithrm.** 17](#_Toc133991058)

**BÁO CÁO THU HOẠCH 2**

1. **Mô hình thuật toán tham lam (Greedy Algorithrm)**
2. **Ý nghĩa và cách hoạt động của thuật toán.**

Thuật toán tham lam (Greedy Algorithm) xây dựng một chiến lược “tham từng miếng”, miếng dễ tham nhất thường là một giải pháp tối ưu cục bộ nhưng lại luôn có mặt trong cấu trúc tối ưu toàn cục. Tiếp tục phát triển chiến lược “tham từng miếng” cho các bước tiếp theo để đạt được kết quả tối ưu toàn cục. Tóm lại, thuật toán tham lam dùng để giải lớp các baì toán tối ưu thỏa mãn hai điều kiện:

• Ở mỗi bước ta luôn tạo ra một lựa chọn tốt nhất tại thời điểm đó.

• Việc liên kết lại kết quả mỗi bước sẽ cho ta kết quả tối ưu toàn cục.

Tổng quát, thuật toán Greedy bao gồm 5 thành phần chính:

1) Một tập các ứng viên (candidate members) mà giải pháp có thể tham lam. 2) Một hàm lựa chọn (selection fuction) để chọn ứng viên tốt nhất cho giải pháp tham lam cục bộ.

3) Một hàm thực thi (feasibility function) được sử dụng để quyết định xem một ứng viên có được dùng để xây dựng lời giải hay không?

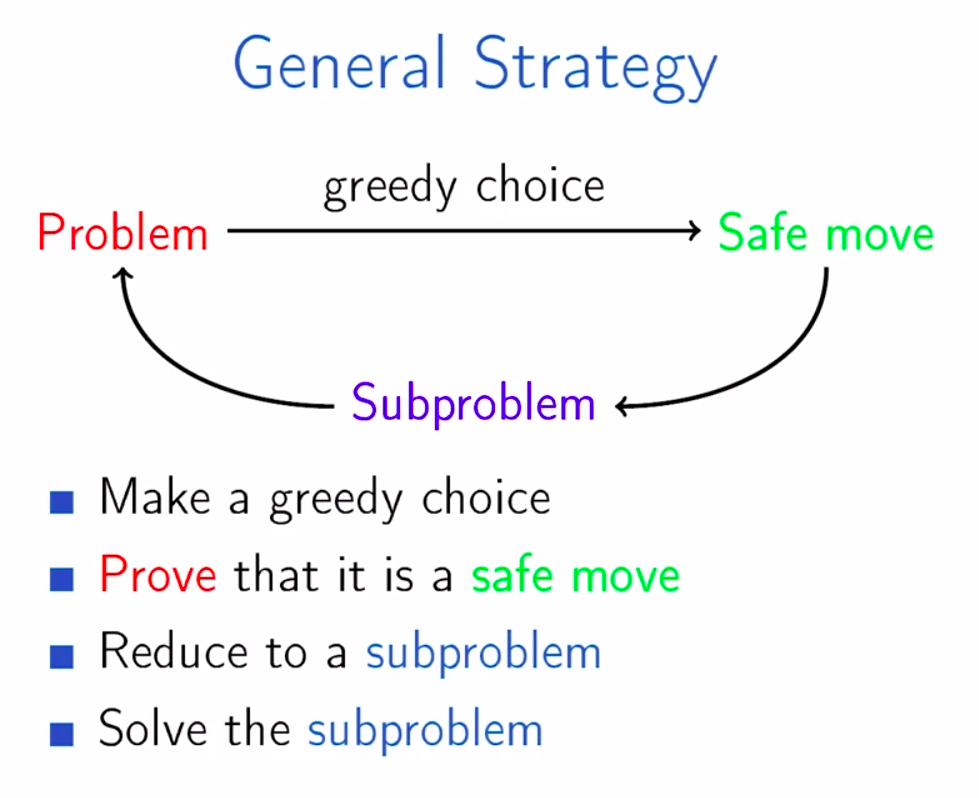
4) Một hàm mục tiêu (objective function) dùng để xác định giá trị của lời giải hoặc một phần của lời giải.

5) Một hàm giải pháp (solution function) dùng để xác định khi nào giải pháp hoàn chỉnh.

Khi sử dụng giải pháp tham lam, hai thành phần quyết định nhất tới quyết định tham lam đó là:

**Lựa chọn tính chất để tham.** Khi chưa tìm được lời giải tối ưu toàn cục nhưng ta biết chắc chắn một giải pháp tối ưu cục bộ dựa vào tính chất cụ thể của mỗi bài toán. Bài toán con tiếp theo cũng thực hiện với tính chất như vậy cho đến khi đạt được giải pháp tối ưu toàn cục.

**Lựa chọn cấu trúc con tối ưu.** Một bài toán được gọi là "có cấu trúc tối ưu" nếu một lời giải tối ưu của bài toán con nằm trong lời giải tối ưu của bài toán lớn hơn. Điều này cho phép ta xác định từng cấu trúc con tối ưu cho mỗi bài toán con, sau đó phát triển các bài toán con cho đến bài toán con cuối cùng.



**Hướng tiếp cận của thuật toán tham lam.**

**1)** Cách tiếp cận tham lam đưa ra các lựa chọn tối ưu cục bộ ở mỗi bước với hy vọng tìm được một tối ưu toàn cục.

**2)** Cách tiếp cận tham lam không nhất thiết phải xem xét hậu quả tương lai của sự lựa chọn hiện tại.

**3)** Cách tiếp cận tham lam rất hữu ích để giải quyết các vấn đề trong đó việc đưa ra các lựa chọn tối ưu cục bộ ở mỗi bước dẫn đến một tối ưu toàn cầu.

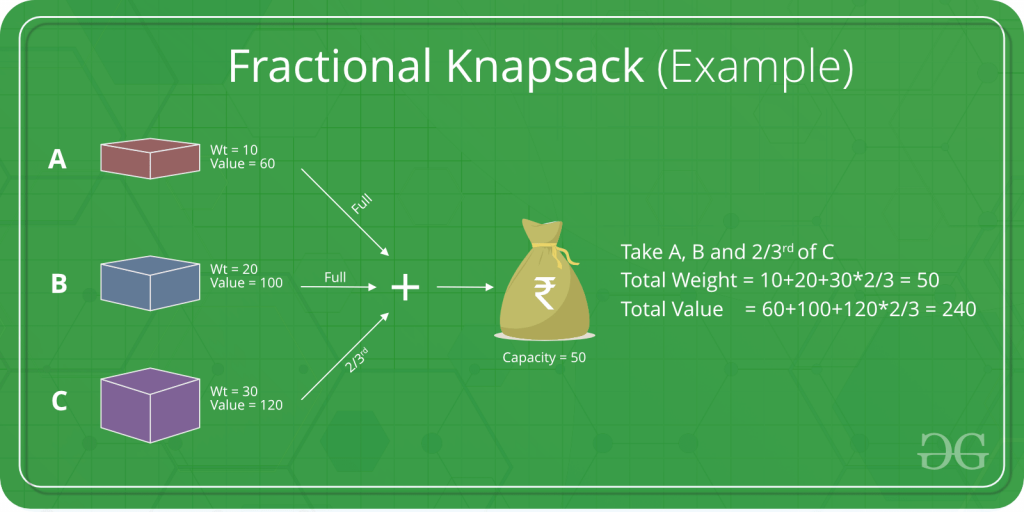
**4)** Cách tiếp cận tham lam thường nhanh hơn và đơn giản hơn quy hoạch động.

1. **Ví dụ**

Ví dụ, xem xét Bài toán Cái túi Phân số.

Cho biết trọng lượng và giá trị của N mặt hàng, dưới dạng **{Value, Weight}**, hãy đặt những mặt hàng này vào một chiếc ba lô có sức chứa W để có được tổng giá trị tối đa trong chiếc ba lô. Trong Fractional Knapsack, chúng ta có thể chia nhỏ các vật phẩm để tối đa hóa tổng giá trị của chiếc ba lô.

Chiến lược tối ưu cục bộ là chọn mặt hàng có tỷ lệ giá trị so với trọng lượng tối đa. Chiến lược này cũng dẫn đến một giải pháp tối ưu toàn cầu vì chúng ta được phép lấy các phân số của một mục.



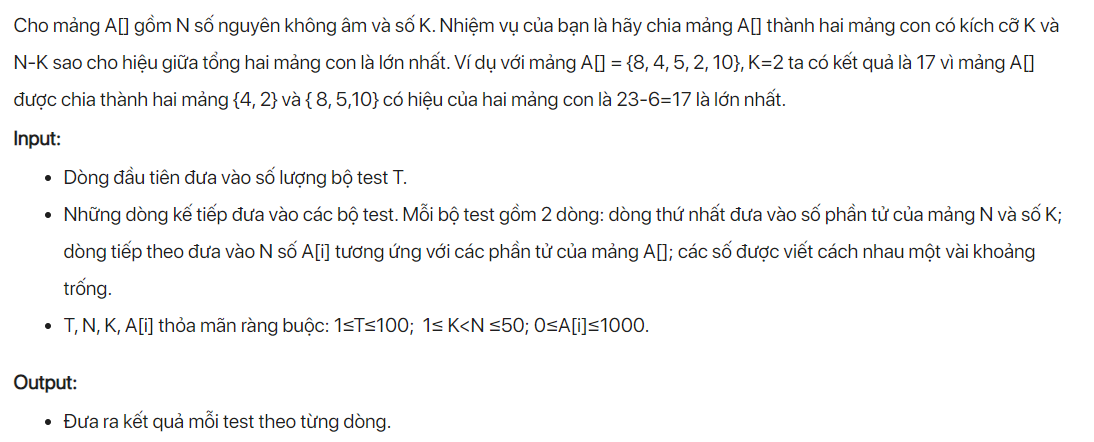
1. **Ứng dụng của thuật toán**
2. [Activity Selection Problem](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)
3. [Job Sequencing Problem](https://www.geeksforgeeks.org/job-sequencing-problem-set-1-greedy-algorithm/)
4. [Huffman Coding](https://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/)
5. [Huffman Decoding](https://www.geeksforgeeks.org/huffman-decoding/)
6. [Water Connection Problem](https://www.geeksforgeeks.org/water-connection-problem/)
7. [Minimum Swaps for Bracket Balancing](https://www.geeksforgeeks.org/minimum-swaps-bracket-balancing/)
8. [Egyptian Fraction](https://www.geeksforgeeks.org/greedy-algorithm-egyptian-fraction/)
9. [Policemen catch thieves](https://www.geeksforgeeks.org/policemen-catch-thieves/)
10. [Fitting Shelves Problem](https://www.geeksforgeeks.org/fitting-shelves-problem/)
11. [Assign Mice to Holes](https://www.geeksforgeeks.org/assign-mice-holes/)
12. **Đánh giá thuật toán**

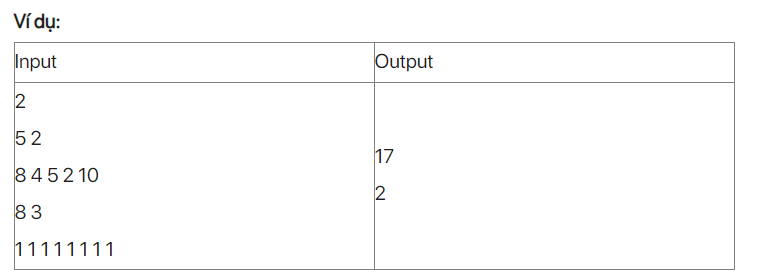
* **Độ phức tạp thời gian:** O(nlogn) hoặc O(n) tùy vào bài toán.
* **Độ phức tạp không gian:** O(1) hoặc O(n) tùy vào bài toán.
* **Tối ưu:** Có thể hoặc không thể đưa ra được giải pháp tối ưu.
* **Ưu điểm:** Thời gian thực hiện nhanh**.**
* **Nhược điểm:** Có thể không tìm dược lời giải tối ưu của bài toán.

1. **CODE PTIT**

**Bài 1: DSA03005 – Chia mảng thành 2 mảng có tổng lớn nhất**

* **Đề bài:**

****

****

* **Phân tích bài toán.**
* Tham lam theo giá trị của các phần tử.
* Đầu tiên chúng ta sắp xếp lại mảng theo thứ tự tăng dần.
* Chúng ta sẽ chia mảng ban đầu thành 2 mảng con có độ dài là k và n-k là 2 mảng có tổng giá trị lớn nhất Max[] và mảng có tổng giá trị bé nhất Min[].
* Với mỗi bước thực hiện chúng ta cố gắng lấy ra các phần tử lần lượt có giá trị lớn nhất trong mảng để chuyển sang cho mảng có tổng giá trị lớn nhất Max[] khi đó sẽ còn lại n-k phần từ nhỏ nhất thuộc về mảng Min[].
* Tính tổng các phần tử trong mảng Max[] – tổng phần tử trong mảng Min[].

**Nhận xét:** **Đối với cách thực hiện này chúng ta cần sắp xếp lại mảng để có thể tham lam các giá trị lớn nhất của mảng. Cách này có thể đưa ra được lời giải tối ưu cho bài toán với thời gian xử lý là n.**

* **Chuyển sang code:** [**https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/43.Hieu\_Lon\_Nhat.cpp**](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/43.Hieu_Lon_Nhat.cpp)

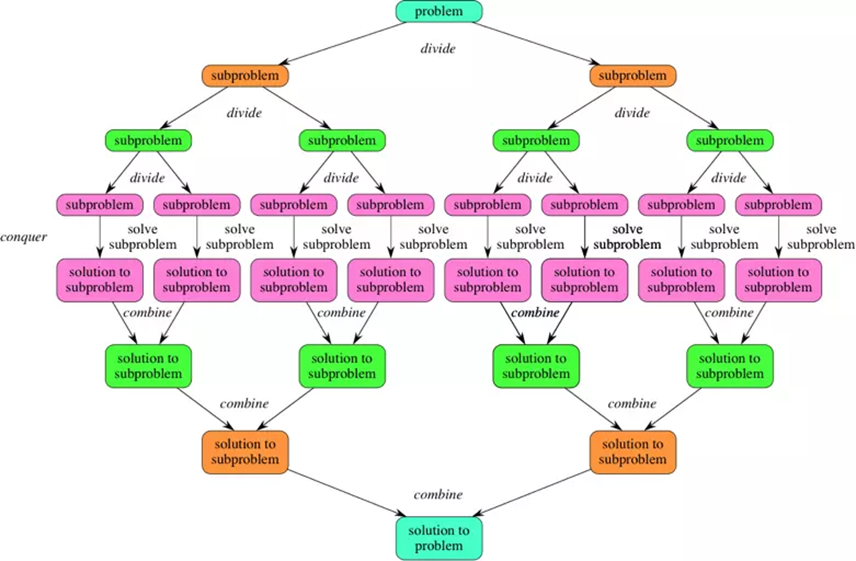
1. **Mô hình thuật toán chia và trị (Devide and conquer Algorithm)**
2. **Ý nghĩa và cách hoạt động của thuật toán.**

Thuật toán chia để trị (Devide and Conquer) dùng để giải lớp các bài toán có thể thực hiện được thông qua ba bước:

• Bước chia (Devide). Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.

• Bước trị (Conquer). Giải các bài toán con đã chia ở bước trước đó. Thông thường các bài toán con chỉ khác nhau về dữ liệu vào nên ta có thể thực hiện bằng một thủ tục đệ qui.

• Bước tổng hợp (Combine). Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn.



**Mã giả:**

**DAC(a, i, j)**

**{**

**if(small(a, i, j))**

**return(Solution(a, i, j))**

**else**

**mid = divide(a, i, j)**

**b = DAC(a, i, mid)**

**c = DAC(a, mid+1, j)**

**d = combine(b, c)**

**return(d)**

**}**

1. **Ví dụ**

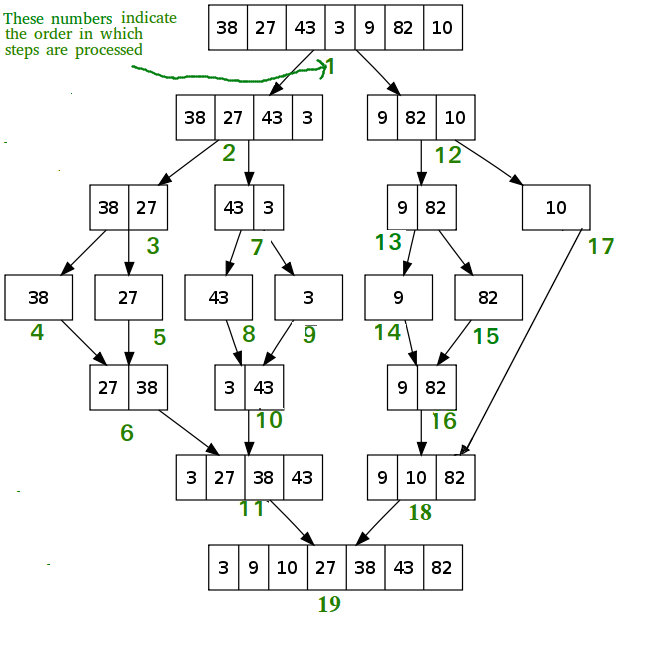
Xét bài toán Merge Sort:

Merge sort được định nghĩa là một thuật toán sắp xếp hoạt động bằng cách chia một mảng thành các mảng con nhỏ hơn, sắp xếp từng mảng con, sau đó hợp nhất các mảng con đã sắp xếp lại với nhau để tạo thành mảng được sắp xếp cuối cùng.

Nói một cách đơn giản, chúng ta có thể nói rằng quá trình merge sort là chia mảng thành hai nửa, sắp xếp từng nửa và sau đó hợp nhất các nửa đã sắp xếp lại với nhau. Quá trình này được lặp lại cho đến khi toàn bộ mảng được sắp xếp.

Hãy coi nó như một thuật toán đệ quy liên tục chia mảng làm đôi cho đến khi không thể chia tiếp nữa. Điều này có nghĩa là nếu mảng trở nên trống hoặc chỉ còn lại một phần tử, phép chia sẽ dừng lại, tức là trường hợp cơ sở để dừng đệ quy. Nếu mảng có nhiều phần tử, hãy chia mảng thành hai nửa và gọi đệ quy sắp xếp hợp nhất trên mỗi nửa. Cuối cùng, khi cả hai nửa được sắp xếp, thao tác hợp nhất sẽ được áp dụng. Hoạt động hợp nhất là quá trình lấy hai mảng được sắp xếp nhỏ hơn và kết hợp chúng để cuối cùng tạo thành một mảng lớn hơn.

Để biết chức năng của sắp xếp hợp nhất, hãy xem xét một mảng arr[] = {38, 27, 43, 3, 9, 82, 10}



**Thuật toán:**

*step 1: start*

*step 2: declare array and left, right, mid variable*

*step 3: perform merge function.  
    if left > right  
        return  
    mid= (left+right)/2  
    mergesort(array, left, mid)  
    mergesort(array, mid+1, right)  
    merge(array, left, mid, right)*

*step 4: Stop*

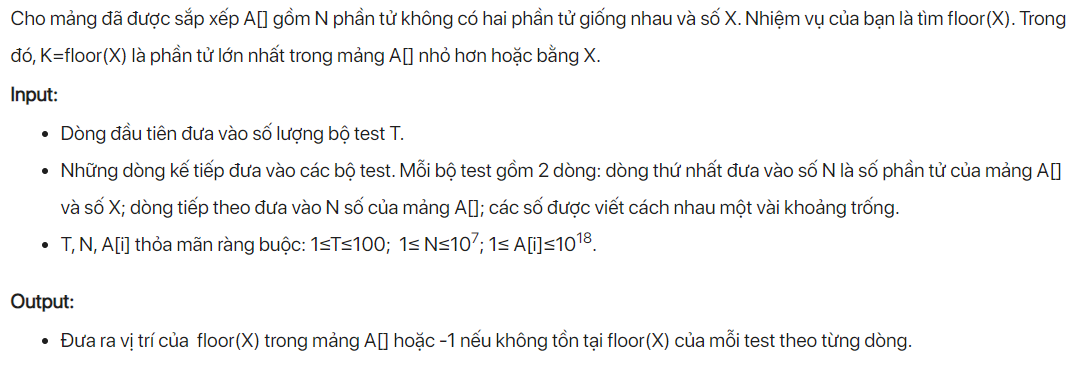
1. **Ứng dụng của thuật toán**
2. [Binary Search](https://www.geeksforgeeks.org/binary-search/)
3. [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)
4. [Quick Sort](https://www.geeksforgeeks.org/quick-sort/)
5. [Calculate pow(x, n)](https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/)
6. [Karatsuba algorithm for fast multiplication](https://www.geeksforgeeks.org/divide-and-conquer-set-2-karatsuba-algorithm-for-fast-multiplication/)
7. [Strassen’s Matrix Multiplication](https://www.geeksforgeeks.org/strassens-matrix-multiplication/)
8. [Convex Hull (Simple Divide and Conquer Algorithm)](https://www.geeksforgeeks.org/convex-hull-simple-divide-conquer-algorithm/)
9. [Quickhull Algorithm for Convex Hull](https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/)
10. **Đánh giá thuật toán**

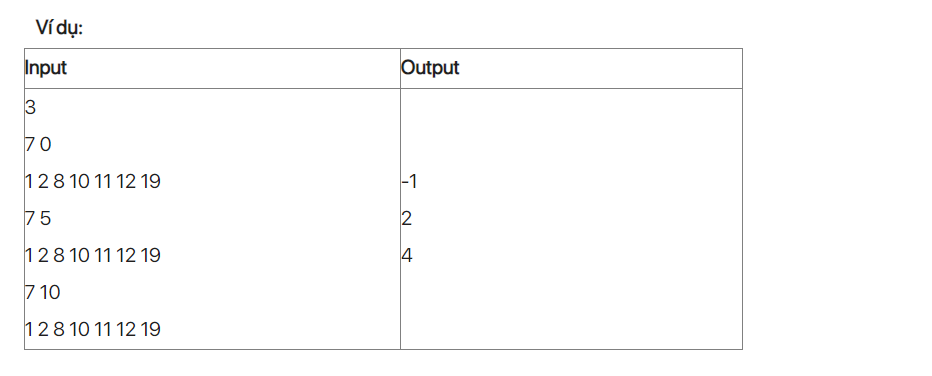
* **Độ phức tạp thời gian:** O(nlogn) or O(n^2) tùy vào từng bài toán.
* **Độ phức tạp không gian:** O(nlogn) or O(n^2) tùy vào từng bài toán.
* **Tối ưu:** Có thể hoặc không thể cung cấp giải pháp tối ưu.
* **Ưu điểm:**
* Giải quyết những bài toán có lời giải phức tạp.
* Nó có thể được sử dụng để giải quyết nhiều vấn đề, bao gồm sắp xếp, tìm kiếm và tối ưu hóa.
* Chia bài toán thành những bài toán con để tối đa thời gian xử lý.
* Giảm độ phức tạo của vấn đề.
* **Nhược điểm:**
* Khó xác định trường hợp hoặc điều kiện dừng cho các cách chia bài toán con.
* Nó có thể không phải là thuật toán hiệu quả nhất cho mọi vấn đề.
* Nó thể yêu cầu nhiều bộ nhớ hơn các thuật toán khác vì nó liên quan đến việc lưu trữ các kết quả trung gian.

1. **CODE PTIT**

**Bài 1: DSA04015 – Tính FLOOR(X)**

* **Đề bài:**

****

****

* **Phân tích bài toán:**
* Mảng đã được sắp xếp nên cách tiếp cận theo phương pháp chia để trị là phương pháp rất hữu dụng.
* Thực hiện như bài toán binary search.
* Chia mảng thành 2 nửa
* Lấy phần tử ở giữa so sánh với phần tử cần tìm kiếm

mid= left + (right – left) / 2

* Nếu 2 phần tử này là bằng nhau -> return vị trí của phần tử hiện tại mid.
* Nếu giá trị tại phần từ mid > giá trị cần tìm: Quay lại tìm ở nửa bên trái của mảng: right = mid – 1;
* Nếu giá trị tại phẩn từ mid < giá trị cần tìm: Quay lại tìm ở nửa bên phải của mảng: left = left + 1;
* Nếu left == 0 return -1 vị tất cả các phần từ đền trong mảng đều > giá trị cần tìm.
* Nếu left != 0 -> return left.

**Nhận xét: vì mảng mảng đã được sắp xếp và độ dài của mảng lớn nên chúng ta không nên sử dụng cách tìm kiếm thông thường. Sử dụng phương pháp chia để trị (tìm kiếm trên cách mảng con) sẽ giảm bớt được thời gian cũng như không gian tìm kiếm.**

**Bài toán con: Tìm giá trị lớn nhất < X trên các mảng con.**

* **Chuyển sang code:** [**https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/DSA04015-Tinh\_Floor(x).cpp**](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/DSA04015-Tinh_Floor(x).cpp)

1. **Mô hình thuật toán qui hoạch động (Dynamic Programming Algorithrm)**
2. **Ý nghĩa và cách hoạt động của thuật toán**

Phương pháp qui hoạch động dùng để giải lớp các bài toán thỏa mãn những điều kiện sau:

• Bài toán lớn cần giải có thể phân rã được thành nhiều bài toán con. Trong đó, sự phối hợp lời giải của các bài toán con cho ta lời giải của bài toán lớn. Bài toán con có lời giải đơn giản được gọi là cơ sở của qui hoạch động. Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn được gọi là công thức truy hồi của qui hoạch động.

• Phải có đủ không gian vật lý lưu trữ lời giải các bài toán con (Bảng phương án của qui hoạch động). Vì qui hoạch động đi giải quyết tất cả các bài toán con, do vậy nếu ta không lưu trữ được lời giải các bài toán con thì không thể phối hợp được lời giải giữa các bài toán con.

• Quá trình giải quyết từ bài toán cơ sở (bài toán con) để tìm ra lời giải bài toán lớn phải được thực hiện sau hữu hạn bước dựa trên bảng phương án của qui hoạch động.

**Cách tiếp cận:**

Quy hoạch động chủ yếu là tối ưu hóa đối với đệ quy đơn giản. Bất cứ nơi nào chúng ta thấy một giải pháp đệ quy có các lệnh gọi lặp lại cho cùng một đầu vào, chúng ta có thể tối ưu hóa giải pháp đó bằng Quy hoạch động. Ý tưởng đơn giản là lưu trữ kết quả của các bài toán con để chúng ta không phải tính toán lại chúng khi cần sau này. Việc tối ưu hóa đơn giản này giúp giảm độ phức tạp về thời gian từ cấp số nhân sang đa thức.

Quy hoạch động là một cách tiếp cận thuật toán từ dưới lên nhằm xây dựng giải pháp cho một vấn đề bằng cách giải các bài toán con của nó theo cách đệ quy.

Quy hoạch động lưu trữ các giải pháp cho các bài toán con và sử dụng lại chúng khi cần thiết để tránh giải các bài toán con giống nhau nhiều lần.

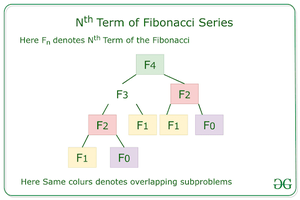
Quy hoạch động rất hữu ích để giải quyết các vấn đề trong đó giải pháp tối ưu có thể thu được bằng cách kết hợp các giải pháp tối ưu cho các bài toán con.

Quy hoạch động thường chậm hơn và phức tạp hơn so với phương pháp tham lam, nhưng nó đảm bảo giải pháp tối ưu.

**Cách giải bài toán Quy hoạch động.**

* **Để giải động một bài toán ta cần kiểm tra hai điều kiện cần:**

**Các bài toán con**: Khi các giải pháp cho các bài toán con giống nhau được lặp đi lặp lại để giải quyết vấn đề thực tế. Vấn đề được cho là có thuộc tính subproblems chồng chéo.



**Thuộc tính cấu trúc con tối ưu( Công thức truy hồi)**: Nếu giải pháp tối ưu của bài toán đã cho có thể đạt được bằng cách sử dụng các giải pháp tối ưu của các bài toán con của nó thì vấn đề được cho là có Thuộc tính cấu trúc con tối ưu.

* **Các bước giải bài toán Quy hoạch động:**
* Xác định xem đó có phải là bài toán Quy Hoạch Động hay không.
* Quyết định một biểu thức trạng thái với các tham số tối thiểu.
* Hình thành các mối quan hệ trạng thái và chuyển tiếp.
* Lập bảng (hoặc ghi nhớ).

1. **Ví dụ**

Xét bài toán cái túi 0/1.

Chúng ta được cung cấp N đồ vật trong đó mỗi mặt hàng có một số trọng lượng và giá trị của nó. Chúng ta cũng được tặng một chiếc túi có sức chứa W, [tức là chiếc túi có thể chứa tối đa trọng lượng W trong đó]. Mục tiêu là đặt các đồ vật vào túi sao cho tổng giá trị của các đồ vật có thể bỏ vào túi là tối đa có thể.

Lưu ý: Hạn chế ở đây là chúng ta có thể cho hoàn toàn một vật phẩm vào túi hoặc không thể cho hết [Không thể cho một phần đồ vật vào túi].

**Example:**

**Input:** N = 3, W = 4, value[] = {1, 2, 3}, weight[] = {4, 5, 1}

**Output**: 3

**Giải thích**: Có hai mặt hàng có trọng lượng nhỏ hơn hoặc bằng 4. Nếu chúng ta chọn mặt hàng có trọng số 4, giá trị có thể có là 1. Và nếu chúng ta chọn mặt hàng có trọng lượng 1, giá trị có thể có là 3. Vì vậy, giá trị tối đa có thể có là 3. Lưu ý rằng chúng tôi không thể đặt cả hai mặt hàng có trọng lượng 4 và 1 cùng nhau vì sức chứa của túi là 4.

**Input**: N = 3, W = 3, value[] = {1, 2, 3}, weight[] = {4, 5, 6}

**Output**: 0

Ý tưởng:

Do các bài toán con được đánh giá lại nên bài toán này có thuộc tính Các bài toán con chồng chéo. Vì vậy, bài toán Cái túi 0/1 có cả hai thuộc tính của bài toán quy hoạch động. Giống như các bài toán Quy hoạch động (DP) điển hình khác, có thể tránh được việc tính toán lại các bài toán con giống nhau bằng cách xây dựng một mảng tạm thời K[][] theo cách từ dưới lên.

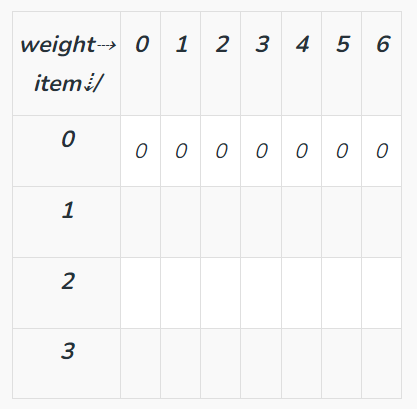
Thực hiện theo các bước dưới đây để giải quyết vấn đề:

* Hãy xem xét các trường hợp tương tự như đã đề cập trong cách tiếp cận đệ quy.
* Trong bảng DP[][], hãy xem xét tất cả các trọng số có thể có từ '1' đến 'W' dưới dạng cột và phần tử có thể được giữ dưới dạng hàng.
* Trạng thái DP[i][j] sẽ biểu thị giá trị lớn nhất của 'j-weight' khi xem xét tất cả các giá trị từ '1 đến i'. Vì vậy, nếu chúng ta xem xét 'wi' (trọng số trong hàng 'ith'), chúng ta có thể điền nó vào tất cả các cột có 'giá trị trọng số > wi'. Bây giờ hai khả năng có thể xảy ra:
* Điền 'wi' vào cột đã cho.
* Không điền 'wi' vào cột đã cho.
* Bây giờ chúng ta phải tận dụng tối đa hai khả năng này,
* Về mặt hình thức, nếu chúng ta không điền trọng số 'ith' vào cột 'jth' thì trạng thái DP[i][j] sẽ giống như DP[i-1][j]
* Nhưng nếu chúng ta điền trọng số thì DP[i][j] sẽ bằng giá trị của (‘wi‘+giá trị của cột có trọng số ‘j-wi’) ở hàng trước.
* Vì vậy, chúng tôi tận dụng tối đa hai khả năng này để điền vào trạng thái hiện tại.

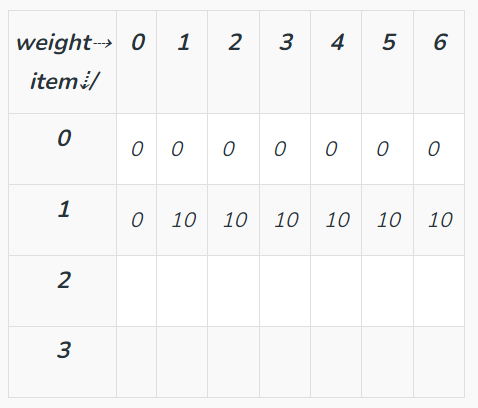
**Dưới đây là hình minh họa của cách tiếp cận trên:**

**Giả sử, weight[] = {1, 2, 3}, value[] = {10, 15, 40}, Capacity = 6**

* Nếu không có phần tử nào được điền, thì giá trị có thể là 0.



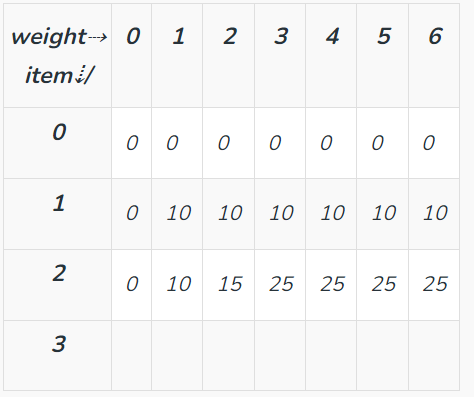
* Để điền mục đầu tiên vào túi: Nếu chúng ta làm theo quy trình đã đề cập ở trên, bảng sẽ như sau



* **Để thêm items thứ 2 vào:**

Khi j = 2 thì giá trị tối đa có thể đạt được là max(10, DP[1][2-2] + 15) = max(10, 15) = 15.

Khi j = 3, thì giá trị tối đa có thể là  max(2 không bỏ, 2 bỏ vào túi) = max(DP[1][3], 15+DP[1][3-2]) = max (10, 25) = 25.



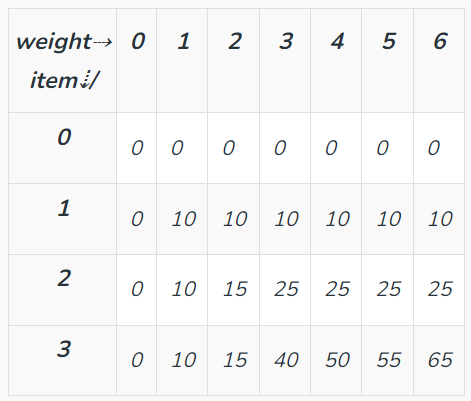
* **Để điền vào item thứ 3:**

Khi j = 3, giá trị tối đa có thể là max(DP[2][3], 40+DP[2][3-3]) = max(25, 40) = 40.

Khi j = 4, giá trị tối đa có thể là max(DP[2][4], 40+DP[2][4-3]) = max(25, 50) = 50.

Khi j = 5, giá trị tối đa có thể đạt được là max(DP[2][5], 40+DP[2][5-3]) = max(25, 55) = 55.

Khi j = 6, giá trị tối đa có thể đạt được là max(DP[2][6], 40+DP[2][6-3]) = max(25, 65) = 65.



**Code:**

// A dynamic programming based

// solution for 0-1 Knapsack problem

#include <bits/stdc++.h>

**using** **namespace** std;

// A utility function that returns

// maximum of two integers

**int** max(**int** a, **int** b) { **return** (a > b) ? a : b; }

// Returns the maximum value that

// can be put in a knapsack of capacity W

**int** knapSack(**int** W, **int** wt[], **int** val[], **int** n)

{

**int** i, w;

    int K[n+1][W+1];

    // Build table K[][] in bottom up manner

**for** (i = 0; i <= n; i++) {

**for** (w = 0; w <= W; w++) {

**if** (i == 0 || w == 0)

                K[i][w] = 0;

**else** **if** (wt[i - 1] <= w)

                K[i][w] = max(val[i - 1]

                                  + K[i - 1][w - wt[i - 1]],

                              K[i - 1][w]);

**else**

                K[i][w] = K[i - 1][w];

        }

    }

**return** K[n][W];

}

// Driver Code

**int** main()

{

**int** profit[] = { 60, 100, 120 };

**int** weight[] = { 10, 20, 30 };

**int** W = 50;

**int** n = **sizeof**(profit) / **sizeof**(profit[0]);

    cout << knapSack(W, weight, profit, n);

**return** 0;

}

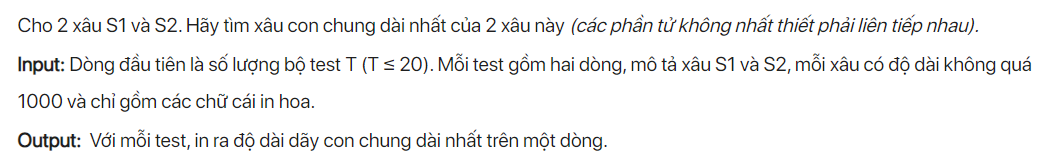
1. **Ứng dụng của thuật toán**
2. [Min Cost Path](https://www.geeksforgeeks.org/min-cost-path-dp-6/)
3. [Subset Sum Problem](https://www.geeksforgeeks.org/subset-sum-problem-dp-25/)
4. [Knapsack problem](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/)
5. [Coin Change](https://www.geeksforgeeks.org/coin-change-dp-7/)
6. [Edit Distance](https://www.geeksforgeeks.org/edit-distance-dp-5/)
7. [Cutting a Rod](https://www.geeksforgeeks.org/cutting-a-rod-dp-13/)
8. [Subset Sum Problem](https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/)
9. [Longest Common Subsequence](https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/)
10. [Matrix chain multiplication](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/)
11. [Count Distinct Subsequences](https://www.geeksforgeeks.org/count-distinct-subsequences/)
12. [Prefix Sum of Matrix (Or 2D Array)](https://www.geeksforgeeks.org/prefix-sum-2d-array/)
13. **Đánh giá thuật toán**

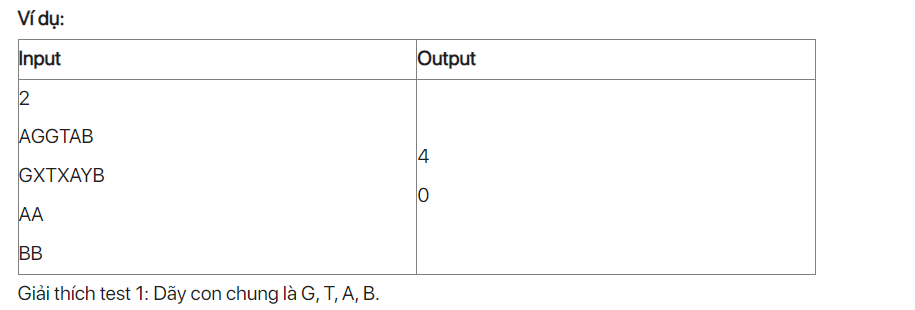
* **Độ phức tạp thời gian:** O(n^2) or O(n^3) tùy vào bài toán.
* **Độ phức tạp không gian:** O(n^2) or O(n^3) tùy vào bài toán.
* **Tối ưu:** Luôn đảm bảo lời giải tối ưu.
* **Ưu điểm:** Luôn cho ra lời giải tối ưu.
* **Nhược điểm:**
* Thời gian và không gian xử lý lớn.
* Khó xác định công thức truy hồi

1. **CODE PTIT**

**Bài 1: DSA05001 – Xâu con chung dài nhất**

* **Đề bài:**

****

****

* **Phân tích bài toán:**

Chúng ta có thể sử dụng các bước sau để triển khai phương pháp Quy hoạch động cho Bài toán này.

Tạo một mảng 2D dp[][] với các hàng và cột bằng độ dài của mỗi chuỗi đầu vào cộng với 1 [số hàng biểu thị chỉ số của S1 và số cột biểu thị chỉ số của S2].

Khởi tạo hàng và cột đầu tiên của mảng dp thành 0.

Lặp lại qua các hàng của mảng dp, bắt đầu từ 1 (giả sử sử dụng trình vòng lặp i).

* Đối với mỗi i, lặp lại tất cả các cột từ j = 1 đến n:
* Nếu S1[i-1] bằng S2[j-1], đặt phần tử hiện tại của mảng dp thành giá trị của phần tử thành (dp[i-1][j-1] + 1).
* Mặt khác, đặt phần tử hiện tại của mảng dp thành giá trị lớn nhất của dp[i-1][j] và dp[i][j-1].

Sau các vòng lặp lồng nhau, phần tử cuối cùng của mảng dp sẽ chứa độ dài của LCS.

**Nhận xét: Bài toán có các bài toán con chồng chéo lên nhau. Dãy con chung lớn nhất của các xâu con có độ dài 1,2,3..n;**

**Cách giải quyết các bài toán con này có lời giải tương tự như giải quyết bài toán lớn.**

**Bài toán con tìm độ dài xâu con dài nhất của các xâu con của 2 xâu ban dầu với độ dài = 1.**

**Sử dụng mảng 2 chiều đề lưu vết các giá trị là độ dài xâu con chung của các xâu con của 2 xâu ban đầu.**

**Công thức truy hồi:**

* **nếu S1[i] = S2[j] thì S[i][j] = S[i-1][j-1] + 1**
* **Ngược lại: S[i][j] = max(S [i-1][j], S[i][j-1]);**
* **Chuyển sang code:** [**https://github.com/namlk173/C-/blob/datastructure\_and\_algorithm/DSA05001-Xau\_Con\_Chung\_Dai\_Nhat.cpp**](https://github.com/namlk173/C-/blob/datastructure_and_algorithm/DSA05001-Xau_Con_Chung_Dai_Nhat.cpp)

1. **So sánh 3 thuật toán Greedy, Divide And Conquer, Dynamic Programming Algorithrm.**

| **Sl.No** | **Greedy Algorithm** | **Divide and conquer** | **Dynamic Programming** |
| --- | --- | --- | --- |
| 1 | Theo cách tiếp cận từ trên xuống | Thực hiện theo cách tiếp cận từ trên xuống (Top-Down) | Thực hiện theo cách tiếp cận từ dưới lên |
| 2 | Dùng để giải bài toán tối ưu | Được sử dụng để giải quyết vấn đề quyết định | Được sử dụng để giải quyết vấn đề tối ưu hóa |
| 3 | Giải pháp tối ưu được tạo ra mà không cần xem lại các giải pháp đã tạo trước đó; Do đó, nó tránh được việc tính toán lại | Giải pháp của vấn đề con được tính đệ quy nhiều lần. | Giải pháp của các vấn đề con được tính toán một lần và được lưu trữ trong một bảng để sử dụng sau này. |
| 4 | Nó có thể hoặc không thể tạo ra một giải pháp tối ưu. | Nó được sử dụng để có được một giải pháp cho vấn đề nhất định, nó không nhằm mục đích cho giải pháp tối ưu | Nó luôn tạo ra giải pháp tối ưu. |
| 5 | Số bước lặp tuần tự | Số lần đệ quy tự nhiên. | Số lần đệ quy tự nhiên.. |
| 6 | **Hiệu quả và nhanh chóng hơn là chi và trị. Ví dụ, việc tìm kiếm đường dẫn ngắn nhất nguồn duy nhất bằng cách sử dụng Algo của Dijkstra mất thời gian O (ElogV)** | kém hiệu quả và chậm hơn. | **hiệu quả hơn nhưng chậm hơn tham lam. Ví dụ, việc tìm kiếm đường đi ngắn nhất nguồn duy nhất bằng cách sử dụng Bellman Ford Algo mất thời gian O (VE).** |
| 7 | Không cần thêm bộ nhớ. | Cần thêm bộ nhớ là bắt buộc. | Cần nhiều bộ nhớ hơn để lưu trữ các vấn đề con để sử dụng sau này. |
| 8 | Examples: [Fractional Knapsack problem](https://www.geeksforgeeks.org/fractional-knapsack-problem/),  [Activity selection problem](https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/),  [Job sequencing problem](https://www.geeksforgeeks.org/job-sequencing-problem/). | Examples: [Merge sort](https://www.geeksforgeeks.org/merge-sort/),  [Quick sort](https://www.geeksforgeeks.org/quick-sort/),  [Strassen’s matrix multiplication](https://www.geeksforgeeks.org/strassens-matrix-multiplication/). | Examples: [0/1 Knapsack](https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/),  [All pair shortest path](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/),  [Matrix-chain multiplication](https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/). |