

# Tầm Quan Trọng Của Thuật Toán

## Tầm Quan Trọng Của Thuật Toán

Bài viết gốc: [The Importance of Algorithms](#) - đăng bởi lbackstrom trên Topcoder [🔗](#)

### Mở đầu

Để hiểu được tầm quan trọng của việc học và hiểu về thuật toán, đầu tiên ta cần định nghĩa chính xác "Thuật toán là gì?". Theo như cuốn sách nổi tiếng *Introduction to Algorithms* (Mở đầu về thuật toán), thì "thuật toán" là "một quá trình tính toán cụ thể, trong đó lấy một hay nhiều giá trị làm đầu vào (**input**) và cho ra một hay nhiều giá trị kết quả (**output**)". Nói cách khác, thuật toán giống như bản đồ chỉ đường giúp ta giải quyết một vấn đề rõ ràng, cụ thể. Ví dụ, nhiều dòng code để tính dãy [Fibonacci](#) [🔗](#) là một cách cài đặt của một thuật toán nào đó. Một hàm để cộng hai số cũng được gọi là một thuật toán, mặc dù nó rất đơn giản.

Một số thuật toán, như thuật để tính dãy Fibonacci, khá trực quan và ta có thể suy ra nhờ vào suy luận logic cũng như kỹ năng giải bài. Tuy nhiên, đối với những thuật toán phức tạp hơn, chúng ta cần học và hiểu, để làm nền tảng cho việc giải quyết những bài toán khó hơn. Trong cuộc sống hàng ngày, những việc tưởng chừng như đơn giản như kiểm tra email hay nghe nhạc đều đòi hỏi những thuật toán rất phức tạp. Trong bài viết này, chúng ta sẽ cùng xem một vài ví dụ về cách áp dụng thuật toán.

### Phân tích thời gian thực hiện thuật toán

Một trong những yếu tố quan trọng nhất của một thuật toán là việc nó chạy nhanh hay chậm. Nghĩ ra một thuật để giải một bài toán là việc dễ, nhưng để nó có ý nghĩa trong thực tế, thì thuật toán phải chạy đủ nhanh. Tốc độ của một thuật tùy thuộc vào tốc độ máy cũng như chi tiết cài đặt, các nhà khoa học máy tính thường đề cập đến **runtime** (thời gian chạy) tương ứng với kích thước input. Ví dụ, nếu input gồm  $N$  số, một thuật toán có thời gian chạy tỉ lệ thuận với  $N^2$  được kí hiệu là  $O(N^2)$ . Kí hiệu này nghĩa là, khi máy tính chạy một chương trình cài đặt thuật toán trên với độ lớn input là  $N$ , chương trình sẽ tốn  $C * N^2$  giây, với  $C$  là một hằng số không phụ thuộc vào kích thước input.

Tuy nhiên, thời gian chạy thực tế của nhiều thuật toán phức tạp còn phụ thuộc nhiều yếu tố khác ngoài độ lớn input. Ví dụ, một thuật toán sắp xếp sẽ chạy nhanh hơn nhiều khi được xử lý một dãy số đã sắp xếp sẵn so với khi phải giải quyết một dãy số lộn xộn. Từ đó sinh ra hai khái niệm: thời gian chạy tối đa và thời gian chạy trung bình. Thời gian chạy tối đa là thời gian thuật toán cần để xử lý input trong trường hợp tồi nhất. Thời gian chạy trung bình là thời gian trung bình cho mọi input có thể xuất hiện. Trong 2 thuật ngữ, thời gian tối đa thường dễ phân tích hơn, nên nó thường được sử dụng để đánh giá các thuật toán. Việc tính toán độ phức tạp nhiều lúc không đơn giản vì ta không thể kiểm tra được hết mọi trường hợp. Bạn có thể tìm đọc thêm về một số kỹ năng tính toán độ phức tạp ở các nguồn khác.




**Ước tính thời gian chạy của các thuật,  $N = 100$**

- $O(\log(N))$ :  $10^{-7}$  giây

- $O(N)$ :  $10^{-6}$  giây
- $O(N * \text{Log}(N))$ :  $10^{-5}$  giây
- $O(N^2)$ :  $10^{-4}$  giây
- $O(N^6)$ : 3 phút
- $O(2^N)$ :  $10^{14}$  năm
- $O(N!)$ :  $10^{142}$  năm




## Sắp xếp

Thuật toán sắp xếp là một thuật toán rất hay được sử dụng trong khoa học máy tính. Cách đơn giản nhất để sắp xếp một nhóm đồ vật là lấy ra món đồ nhỏ nhất và đặt lên đầu. Ta lại lấy ra món đồ nhỏ thứ hai và đặt cạnh món đầu tiên; cứ thế tiếp tục. Đáng tiếc, độ phức tạp của thuật toán này là  $O(N^2)$ , tức thời gian chạy của thuật sẽ tương ứng với bình phương độ lớn input. Nếu cần sắp xếp 1 tỉ số, thuật sẽ cần  $10^{18}$  **câu lệnh máy tính** (instructions). Để dễ hình dung thì một chiếc máy tính hiện đại mỗi giây chỉ giải quyết được gần  $10^9$  câu lệnh, và sẽ cần nhiều năm mới sắp xếp xong 1 tỉ con số nói trên.

May thay, có nhiều thuật toán sắp xếp khác hiệu quả hơn như [Quick Sort](#) , [Heap Sort](#) , [Merge Sort](#) , ... Nhiều thuật có độ phức tạp chỉ  $O(N * \text{Log}(N))$ . Độ phức tạp nhỏ hơn rút ngắn rất nhiều thời gian để sắp xếp 1 tỉ con số, đến mức một chiếc máy tính bình thường nhất cũng thực hiện được trong chưa đến một phút. Thay vì  $10^{18}$  câu lệnh, máy chỉ phải thực hiện  $10^{10}$ .

## Đường đi ngắn nhất


Thuật toán tìm đường đi ngắn nhất giữa hai điểm đã được nghiên cứu từ rất lâu. Ứng dụng của thuật toán này rất nhiều, nhưng tạm thời hãy xét bài toán cơ bản nhất: tìm đường đi ngắn nhất từ A đến B trong một thành phố với một số con đường và giao lộ. Có rất nhiều thuật toán được phát triển để giải quyết các bài kiểu này, với mỗi phương pháp có ưu nhược điểm riêng. Trước khi tìm hiểu về chúng, hãy xem thử "thuật trâu" - xét hết các khả năng - sẽ mất bao lâu để giải. Đáng tiếc, ta sẽ không sống đủ lâu để biết kết quả từ phương pháp này - cho dù A và B đều ở trong một thị trấn nhỏ, bởi độ phức tạp của thuật là  $O(C^N)$ , với  $C$  là hằng số. Dù  $C$  có nhỏ,  $C^N$  sẽ trở nên lớn khủng khiếp ngay cả với các giá trị vừa phải của  $N$ .

Một trong những thuật toán nhanh nhất để giải quyết bài toán này có độ phức tạp  $O(E * \text{Log}(V))$ , với  $E$  là số đường đi,  $V$  là số giao lộ. Để dễ hình dung, thuật toán sẽ mất 2 giây để tìm đường đi ngắn nhất trong thành phố có 10000 giao lộ và 20000 con đường. Thuật toán này - [Dijkstra](#)  - khá phức tạp và đòi hỏi sử dụng cấu trúc dữ liệu [Hàng đợi ưu tiên \(Priority Queue\)](#) . Tuy nhiên có những trường hợp mà ngay cả [Dijkstra](#) cũng trở nên quá chậm (ví dụ đường đi ngắn nhất từ Hà Nội đến Silicon Valley - có đến hàng tỉ giao lộ), các lập trình viên sẽ sử dụng **heuristics**. Heuristics là một hàm xấp xỉ của một tính chất của bài toán. Ví dụ, trong bài toán này, 1 heuristic có thể là khoảng cách đường chim bay giữa 2 điểm. Sử dụng heuristic, chúng ta có thể tìm được lời giải nhanh hơn, ví dụ [thuật toán A\\*](#)  có thể chạy nhanh hơn Dijkstra trong nhiều trường hợp. Phương pháp này không phải lúc nào cũng rút ngắn thời gian chạy của thuật trong trường hợp tệ nhất, nhưng nó hiệu quả với phần lớn trường hợp trong thực tế.


## Thuật toán xấp xỉ

Thế nhưng đôi khi, ngay cả với thuật mạnh nhất, dùng heuristics tốt nhất, chạy trên máy tính nhanh nhất cũng trở nên quá chậm. Đối với các trường hợp này ta buộc phải hi sinh phần nào sự chính xác của kết quả. Thay vì cố đi tìm đường đi ngắn nhất, ta đã có thể thỏa mãn với một con đường không dài hơn 10% so với kết quả tối ưu.

Thực tế, có nhiều bài toán quan trọng mà thuật tối ưu của chúng quá chậm và không sử dụng được. Nhóm bài nổi tiếng nhất cho tính cách này được gọi là NP - non-deterministic polynomial. Khi một bài toán nào đó được cho là NP-hard hay NP-complete, đồng nghĩa không ai biết thuật toán tối ưu của nó. Hơn nữa, nếu ai đó nghĩ ra được thuật tối ưu cho một bài toán NP-complete, thuật đó cũng áp dụng được cho mọi bài toán NP-complete khác.

Một ví dụ nổi tiếng của NP-hard là bài toán **Người bán hàng (Traveling Salesman Problem - TSP)** . Một người bán hàng cần đi đến  $N$  thành phố, và anh ta biết cần bao lâu để đi từ một thành phố đến một thành phố khác. Câu hỏi đặt ra là, nhanh nhất thì mất bao lâu để anh ta đi hết  $N$  thành phố? Bởi thuật toán nhanh nhất cho TSP vẫn quá chậm - nhiều người tin là điều này sẽ luôn đúng - ta tìm đến những phương pháp đủ nhanh mà đưa ra được kết quả gần đúng, thay vì thuật toán tối ưu.


## Thuật toán ngẫu nhiên

Một cách tiếp cận vấn đề khác đó là thử "random" theo một cách nào đó. Tuy không cải thiện tốc độ trong trường hợp xấu nhất, biện pháp này sẽ có hiệu quả với những trường hợp bình thường. **QuickSort**  là một ví dụ điển hình cho việc random như vậy. Khi "suy biến", QuickSort sẽ sắp xếp một dãy trong  $O(N^2)$ , với  $N$  là số phần tử của dãy. Nhưng nếu áp dụng random vào thuật toán, xác suất xảy ra suy biến là cực kì thấp, và độ phức tạp trung bình của QuickSort trở thành  $O(N * \text{Log}(N))$ . Có những thuật có độ phức tạp  $O(N * \text{Log}(N))$  kể cả ở trường hợp xấu, nhưng chúng sẽ chậm hơn ở trường hợp bình thường. Dù độ phức tạp cùng là  $O(N * \text{Log}(N))$ , QuickSort có hằng số  $C$  nhỏ hơn nhiều, tức số phép tính cần thực hiện nhỏ hơn.

Một thuật toán khác dùng random để tìm trung vị của một dãy với độ phức tạp trung bình  $O(N)$ . Đây là cải tiến rất lớn so với việc sắp xếp lại dãy và chọn ra phần tử giữa vốn có độ phức tạp  $O(N * \text{Log}(N))$ . Thêm nữa, cho dù vẫn có các thuật toán tìm trung vị không-random với độ phức tạp  $O(N)$ , thuật toán này vẫn được áp dụng nhiều hơn nhờ cài đặt giản và tốc độ chạy nhanh hơn.

Ý tưởng chính của Thuật toán trung vị này là chọn ngẫu nhiên một số  $X$  bất kì trong dãy, và đếm xem có bao nhiêu số nhỏ hơn nó. Giả sử dãy có  $N$  số, và có  $K$  số nhỏ hơn hoặc bằng  $X$ . Nếu  $K < N/2$ , vậy ta biết trung vị sẽ là số thứ  $(N/2 - K)$  lớn hơn  $X$ . Ta bỏ đi  $K$  số nhỏ hơn hoặc bằng  $X$ . Bây giờ ta phải tìm số nhỏ thứ  $(N/2 - K)$ , thay vì số trung vị. Ta tiếp tục random một số  $X$  khác, và lặp lại.


## Compression - Nén

Một lớp khác của thuật toán sẽ xử lí những vấn đề như **data compression**  (nén dữ liệu). Loại thuật toán này không có output xác định mà thay vào đó cố gắng tối ưu ở những tiêu chí khác. Với data compression, thuật (như LZW) sẽ cố gắng để cho output nhỏ nhất có thể, và vẫn có khả năng phục hồi lại trạng thái ban đầu. Những thuật toán kiểu này cho ra kết quả có thể chấp nhận được, nhưng không phải tối ưu nhất. Ví dụ thuật toán của JPG và MP3, dù khiến dữ liệu bị giảm chất lượng đi đôi chút, sẽ tạo ra những files nhỏ hơn files gốc rất nhiều. Files MP3 không cố gắng giữ lại mọi âm thanh của bài hát mà ghi vừa đủ thông tin để chất lượng vừa tốt, dung lượng files lại nhỏ. Files JPG cũng dựa trên ý tưởng tương tự.

## Tầm quan trọng của thuật toán



Với một lập trình viên, hay một nhà khoa học máy tính, hiểu rõ về các thuật toán vừa nêu là rất quan trọng để có thể áp dụng thực tiễn. Nếu ta viết một phần mềm, ta sẽ phải đánh giá được phần mềm đó sẽ hoạt động nhanh chậm ra sao. Những đánh giá như vậy sẽ kém chính xác hơn nhiều nếu ta không có hiểu biết về thời gian chạy hay độ phức tạp. Thêm nữa, hiểu biết về thuật toán của những gì ta đang làm sẽ giúp ta dự đoán những trường hợp đặc biệt khiến phần mềm chạy chậm đi hay xảy ra lỗi.

Tất nhiên, ta sẽ thường xuyên gặp những bài toán chưa được nghiên cứu trước đó. Lúc này ta phải tự nghĩ ra thuật mới, hoặc áp dụng thuật cũ một cách sáng tạo hơn. Càng có kiến thức về thuật toán, ta càng có khả năng giải quyết thành công vấn đề. Trong nhiều trường hợp, một vấn đề mới có thể được đưa về một vấn đề cũ hơn mà không cần quá nhiều sức lực, với điều kiện ta phải có kiến thức đủ sâu về vấn đề cũ này.

Để minh họa, hãy tưởng tượng về công việc của một "switch" internet. Một switch có  $N$  sợi dây cáp, và sẽ nhận các gói data từ các sợi cáp này. Switch sẽ phải phân tích các gói data này rồi trả chúng về đúng dây cáp cũ. Cái switch này cũng giống như một chiếc máy tính, làm việc dựa trên các xung nhịp với các bước rành mạch; các gói dữ liệu được gửi ra ở các quãng nghỉ - intervals - thay vì liên tục. Ở một switch nhanh, chúng ta cần gửi các gói dữ liệu này càng nhanh càng tốt ở mỗi quãng nghỉ để chúng không ứ đọng và bị bỏ qua. Vậy mục tiêu của thuật toán là gửi càng nhiều gói data càng tốt ở mỗi quãng nghỉ, và gói nào đến trước thì được chuyển đi trước. Hóa ra, một thuật toán có tên "**stable matching**  " có liên hệ trực tiếp tới vấn đề này và hoàn toàn có thể áp dụng, dù nhìn sơ mỗi tương quan giữa vấn đề và thuật toán có vẻ không rõ ràng. Chỉ có nền tảng thuật toán vững vàng mới giúp ta đi đến lời giải cho những trường hợp như vậy.

## Một vài ví dụ thực tế

Có rất nhiều ví dụ cho thấy các vấn đề thực tế đòi hỏi hiểu biết về thuật toán. Gần như mọi thứ bạn đang làm với máy tính được dựa trên một thuật toán nào đó mà có người phải rất vất vả mới tìm ra. Cho dù là ứng dụng đơn giản nhất của máy tính hiện đại cũng cần đến thuật toán để quản lí dữ bộ nhớ và truy xuất thông tin từ hard drive (ổ cứng).

Có rất nhiều ví dụ trong đời sống, ở đây tôi sẽ giới thiệu 2 bài toán thực tế có cùng độ khó với các bài ở TopCoder. Bài đầu là **Maximum Flow Problem (Luồng Cực Đại)** , và bài sau có liên quan đến **Dynamic Programming (Quy Hoạch Động)**  - một kĩ thuật giải quyết được những vấn đề tưởng như không thể với tốc độ cao.

## Maximum Flow - Luồng cực đại

Bài toán luồng cực đại liên quan đến bài toán tìm đường đi ngắn nhất đã được nói ở trên. Vào những năm 1950, bài toán lần đầu được nghiên cứu, để áp dụng với mạng lưới đường sắt của Liên Xô. Mỹ muốn biết Liên Xô có thể tiếp tế cho các nước vệ tinh ở Đông Âu thông qua mạng lưới đường sắt nhanh như thế nào.

Thêm vào đó, Mỹ còn muốn biết tuyến đường nào có thể bị phá hoại dễ dàng nhất nhằm chia cắt các nước này khỏi Liên Bang. Hóa ra, 2 vấn đề này liên quan mật thiết, và giải quyết được bài toán về tiếp tế sẽ giải quyết luôn vấn đề thứ hai.

Thuật toán hiệu quả đầu tiên cho bài toán tìm luồng cực đại được phát triển bởi Ford và Fulkerson; thuật toán được đặt tên 2 nhà khoa học máy tính này và đã trở thành một trong những thuật toán nổi tiếng nhất ngành. Trong 50 năm qua, một số cải tiến đã được áp dụng để thuật nhanh hơn, nhiều trong số đó vô cùng tinh tế.

Từ lúc bài toán được nghiên cứu, có nhiều ứng dụng đã được đưa ra. Ví dụ trong mạng máy tính, ta cần truyền thật nhiều dữ liệu từ một điểm đến một điểm khác. Trong kinh doanh, thuật toán này được áp dụng trong **nghiên cứu vận hành (operations research)**. Ví dụ, nếu bạn có  $N$  nhân viên và  $N$  công việc cần làm, nhưng không phải ai cũng làm được tất cả mọi công việc, thuật toán tìm luồng cực đại sẽ chỉ ra cách giao việc để các công việc đều hoàn thành, miễn là có phương án khả thi.

## Sequence Comparison - So sánh chuỗi

Nhiều coder đi làm cả đời mà không từng phải cài một thuật toán quy hoạch động nào. Thế nhưng quy hoạch động là cần thiết cho nhiều thuật toán quan trọng. Một thuật toán có lẽ nhiều người đã từng sử dụng qua mà không biết, là tìm khoảng cách của 2 chuỗi. Cụ thể hơn, tính toán xem sau bao nhiêu lần thêm, xóa hay sửa thì xâu A sẽ trở thành xâu B.

Ví dụ, có 2 xâu "AABAA" và "AAAB". Để chuyển xâu đầu thành xâu sau, đơn giản nhất là xóa kí tự 'B' ở giữa rồi chuyển 'A' cuối xâu thành 'B'. Thuật toán này có rất nhiều ứng dụng, ví dụ như trong các vấn đề liên quan đến DNA hay chống đạo văn. Với các lập trình viên, thuật toán này thường được dùng trong việc so sánh 2 phiên bản source code (mã nguồn) của cùng 1 file. Nếu các phần tử của chuỗi là các dòng của file, thuật sẽ cho ta biết dòng code nào bị xóa, dòng nào bị thêm vào hay sửa đi trong các phiên bản đó.

Không có quy hoạch động, ta sẽ phải xét các trường hợp theo cấp số mũ để biến một xâu thành xâu khác. Với quy hoạch động, bài toán này được giải quyết với độ phức tạp chỉ  $O(N * M)$ , trong đó  $N$  và  $M$  là số phần tử của mỗi xâu.

## Kết luận

Số thuật toán khác nhau mà con người học cũng nhiều như số bài toán khác nhau mà ta cần giải. Thế nhưng khả năng cao là bài toán bạn đang cố gắng giải có liên quan đến một bài toán khác, theo một cách nào đó. Có một vốn hiểu biết sâu và rộng về các thuật toán sẽ giúp bạn chọn lựa được hướng đi đúng và áp dụng thành công. Khi nghiên cứu thuật toán, nhiều bài toán nhìn có vẻ không thực tế, nhưng kĩ năng giải quyết đó lại được áp dụng trong những bài toán mà chúng ta gặp hàng ngày.

Được cung cấp bởi [Wiki.js](#)