TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÀI TẬP LỚN/ĐỒ ÁN CUỐI KÌ MÔN**

**Phân Tích Và Thiết Kế Giải Thuật**

**Final Report**

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

*Người thực hiện*: **ĐINH HỒNG HÀ – 518H0171**

**PHÙNG QUỐC ĐẠT – 518H0482**

Lớp **: 18H50205**

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 TẬP LỚN/ĐỒ ÁN CUỐI KÌ MÔN**

**Phân Tích Và Thiết Kế Giải Thuật**

**Final Report**

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

*Người thực hiện*: **ĐINH HỒNG HÀ – 518H0171**

**PHÙNG QUỐC ĐẠT – 518H0482**

Lớp **: 18H50205**

Khoá  **: 22**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

LỜI CẢM ƠN

Em cảm ơn thầy rất nhiều vì đã rất nhiệt tình giúp đỡ chỉ bảo với bọn em trong quá trình học tập. Những bài học thầy giảng cho bọn em thật bổ ích , thầy đã tận tình chỉ những lỗi cho bọn em biết để sửa chửa cảm ơn thầy vì điều đó. Em xin chúc thầy luôn thành công trong công việc và cuộc sống và thực hiện tâm huyết của mình với công việc.

**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 / chúng tôi và được sự hướng dẫn của TS 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 11 tháng 10 năm 2020*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Đinh Hồng Hà*

*Phùng Quốc Đạt*

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

**Phần xác nhận của GV 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)

TÓM TẮT

* Chúng ta sẽ đi nghiên cứu trong 2 phần :
  + Phần 1: Giới thiệu về phân tích và thiết kế giải thuật
  + Phần 2 : Chúng ta sẽ làm về phần đã học , được học và tự học
* Về phần đã học và được học , chúng ta sẽ nghiên cứu và giải quyết về các kỹ thuật:
  + Brute-force
  + Divide-and-conquer
  + Greedy Algorithms
  + Dynamic Programming.
* Về phần tự học , chúng ta sẽ giới thiệu và hiểu về kỹ thuật Backtracking.

MỤC LỤC

[LỜI CẢM ƠN i](#_Toc54960633)

[PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN iii](#_Toc54960634)

[TÓM TẮT iv](#_Toc54960635)

[MỤC LỤC 1](#_Toc54960636)

[CHƯƠNG 1- GIỚI THIỆU 4](#_Toc54960637)

[1.1 Thuật toán là gì 4](#_Toc54960638)

[1.2 Các nguyên tắc cơ bản giải quyết vấn đề về thuật toán 4](#_Toc54960639)

[1.2.1 Hiểu vấn đề 4](#_Toc54960640)

[1.2.2 Xác minh khả năng của thiết bị tính toán 4](#_Toc54960641)

[1.2.3 Lựa chọn giữa giải quyết vấn đề chính xác và gần đúng 4](#_Toc54960642)

[1.2.4 Kỹ thuật thiết kế thuật toán 5](#_Toc54960643)

[1.2.5 Thiết kế thuật toán và cấu trúc dữ liệu 5](#_Toc54960644)

[1.2.6 Phương pháp chỉ định một thuật toán 5](#_Toc54960645)

[1.2.7 Chứng minh tính đúng đắn của thuật toán 6](#_Toc54960646)

[1.2.8 Phân tích một thuật toán 7](#_Toc54960647)

[1.3 Important Problem Types 7](#_Toc54960648)

[1.3.1 Sorting 8](#_Toc54960649)

[1.3.2 Searching 8](#_Toc54960650)

[1.3.3 String Processing 8](#_Toc54960651)

[1.3.4 Graph Problems 8](#_Toc54960652)

[1.3.5 Geometric Problems 9](#_Toc54960653)

[1.3.6 Numerical Problems 9](#_Toc54960654)

[1.4 Cấu trúc dữ liệu cơ bản 9](#_Toc54960655)

[1.4.1 Cấu trúc dữ liệu tuyến tính 10](#_Toc54960656)

[1.4.2 Đồ thị 10](#_Toc54960657)

[1.4.3 Trees 11](#_Toc54960658)

[CHƯƠNG 2- BRUTE - FORCE 12](#_Toc54960659)

[2.1 Khái niệm: 12](#_Toc54960660)

[2.2 Selection Sort và Bubble Sort: 12](#_Toc54960661)

[2.2.1 Selection Sort: 12](#_Toc54960662)

[2.2.2 Bubble Sort: 15](#_Toc54960663)

[2.3 Search: 18](#_Toc54960664)

[2.3.1 Sequential Search. 18](#_Toc54960665)

[2.4 Kết luận: 22](#_Toc54960666)

[CHƯƠNG 3 – DIVIDE AND CONQUER 23](#_Toc54960667)

[3.1 Merge Sort 23](#_Toc54960668)

[3.2 Quick Sort 28](#_Toc54960669)

[3.3 Binary Tree Traversals 34](#_Toc54960670)

[3.4 Kết luận: 37](#_Toc54960671)

[CHƯƠNG 4- GREEDY ALGORITHMS 38](#_Toc54960672)

[4.1 Tìm cây khung nhỏ nhất 39](#_Toc54960673)

[4.1.1 Thuật toán Prim 39](#_Toc54960674)

[4.1.2 Kruskal’s Algorithm 45](#_Toc54960675)

[4.2 Dijkstra’s Algorithm 53](#_Toc54960676)

[4.3 Kết luận 57](#_Toc54960677)

[CHƯƠNG 5 – DYNAMIC PROGRAMMING 58](#_Toc54960678)

[5.1 Three Basic Examples 58](#_Toc54960679)

[5.1.1 Coin Row Problem 58](#_Toc54960680)

[5.1.2 Change Making Problem: 62](#_Toc54960681)

[5.1.3 Coin Collecting Problem: 66](#_Toc54960682)

[5.2 Kết Luận 70](#_Toc54960683)

[CHƯƠNG 6 - BACKTRACKING 71](#_Toc54960684)

[6.1 Queen Problem 72](#_Toc54960685)

[6.2 Hamiltonian Circuit Problem 74](#_Toc54960686)

[6.3 Subset Sum Problem 75](#_Toc54960687)

[6.4 General Remarks 76](#_Toc54960688)

[BÁO CÁO MỖI TUẦN 79](#_Toc54960689)

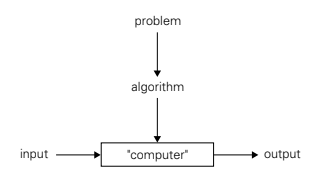
CHƯƠNG 1- GIỚI THIỆU

Tại sao bạn cần nghiên cứu các thuật toán? Nếu bạn định trở thành một chuyên gia máy tính, có cả lý do thực tế và lý thuyết để nghiên cứu thuật toán. Từ quan điểm thực tế, bạn phải biết một bộ tiêu chuẩn các thuật toán quan trọng từ các lĩnh vực máy tính khác nhau. Ngoài ra, bạn sẽ có thể thiết kế các thuật toán mới và phân tích hiệu quả của chúng.

* 1. Thuật toán là gì

Thuật toán là một chuỗi các hướng dẫn rõ ràng để giải quyết một vấn đề, tức là để có được đầu ra bắt buộc cho bất kỳ đầu vào hợp pháp nào trong một khoảng thời gian hữu hạn.

Định nghĩa này có thể được minh họa bằng một sơ đồ đơn giản :



1.2 Các nguyên tắc cơ bản giải quyết vấn đề về thuật toán

1.2.1 Hiểu vấn đề

Từ góc độ thực tế, điều đầu tiên bạn cần làm trước khi thiết kế một thuật toán là hiểu hoàn toàn vấn đề được đưa ra. Đọc kỹ mô tả vấn đề và đặt câu hỏi nếu bạn có bất kỳ nghi ngờ nào về vấn đề, làm một vài ví dụ nhỏ bằng tay, suy nghĩ về những trường hợp đặc biệt, và đặt câu hỏi nếu cần.

1.2.2 Xác minh khả năng của thiết bị tính toán

Khi bạn hoàn toàn hiểu một vấn đề, bạn cần xác định khả năng của thiết bị tính toán mà thuật toán hướng tới. Phần lớn các thuật toán được sử dụng ngày nay vẫn được lập trình chặt chẽ cho máy tính .

1.2.3 Lựa chọn giữa giải quyết vấn đề chính xác và gần đúng

Quyết định chính tiếp theo là lựa chọn giữa giải quyết vấn đề chính xác hoặc giải quyết nó một cách gần đúng. Trong trường hợp trước đây, một thuật toán được gọi là một thuật toán chính xác; trong trường hợp thứ hai, một thuật toán được gọi là thuật toán xấp xỉ. Tại sao một người sẽ chọn một thuật toán xấp xỉ? Đầu tiên, có những vấn đề quan trọng không thể được giải quyết chính xác đối với hầu hết các trường hợp của chúng; các ví dụ bao gồm trích xuất căn bậc hai, giải câu lệnh phi tuyến và đánh giá định nghĩa

tích phân inite. Thứ hai, các thuật toán có sẵn để giải quyết chính xác một vấn đề có thể chậm đến mức không thể chấp nhận được vì độ phức tạp nội tại của vấn đề. Điều này đặc biệt xảy ra đối với nhiều vấn đề liên quan đến một số lượng rất lớn các lựa chọn. Thứ ba, thuật toán xấp xỉ có thể là một phần của thuật toán phức tạp hơn giúp giải quyết chính xác vấn đề.

1.2.4 Kỹ thuật thiết kế thuật toán

Kỹ thuật thiết kế thuật toán ( “chiến lược” hoặc “mô hình”) là một cách tiếp cận chung để giải quyết các vấn đề theo thuật toán có thể áp dụng cho nhiều vấn đề từ các lĩnh vực máy tính khác nhau.

1.2.5 Thiết kế thuật toán và cấu trúc dữ liệu

Mặc dù các kỹ thuật thiết kế thuật toán cung cấp một tập hợp các ứng dụng mạnh mẽ hướng dẫn giải quyết vấn đề theo thuật toán, thiết kế một thuật toán cho một vấn đề cụ thể vẫn có thể là một nhiệm vụ đầy thách thức. Một số kỹ thuật thiết kế có thể đơn giản là không thể áp dụng cho vấn đề được đề cập. Đôi khi, một số kỹ thuật cần được kết hợp với nhau và có những thuật toán khó xác định là ứng dụng của các kỹ thuật thiết kế đã biết. Ngay cả khi một kỹ thuật thiết kế cụ thể là phù hợp, việc tạo ra một thuật toán thường đòi hỏi sự khéo léo không hề nhỏ của người thiết kế thuật toán. Với thực hành, cả hai nhiệm vụ lựa chọn trong số các kỹ thuật chung và áp dụng chúng trở nên dễ dàng hơn, nhưng chúng hiếm khi dễ dàng.

1.2.6 Phương pháp chỉ định một thuật toán

Sử dụng một ngôn ngữ tự nhiên có một sức hấp dẫn rõ ràng; tuy nhiên, đặc tính xung quanh vốn có của bất kỳ ngôn ngữ tự nhiên nào khiến cho việc mô tả các thuật toán ngắn gọn và rõ ràng trở nên khó khăn một cách đáng ngạc nhiên. Tuy nhiên, có thể làm được điều này là một kỹ năng quan trọng mà bạn nên cố gắng phát triển trong quá trình học thuật toán.

Mã giả là một hỗn hợp của ngôn ngữ tự nhiên và các cấu trúc giống như ngôn ngữ lập trình. Mã giả thường chính xác hơn ngôn ngữ tự nhiên và việc sử dụng nó thường mang lại các mô tả thuật toán ngắn gọn hơn.

Hiện đại của máy tính vẫn chưa đạt đến mức mô tả thuật toán có thể là bằng ngôn ngữ tự nhiên hoặc mã giả có thể được đưa trực tiếp vào máy tính điện tử. Thay vào đó, nó cần được chuyển đổi thành chương trình máy tính được viết bằng một ngôn ngữ máy tính cụ thể. Chúng ta có thể xem một chương trình như vậy là một cách khác để chỉ định thuật toán, Mặc dù nên coi nó như là cách triển khai của thuật toán.

1.2.7 Chứng minh tính đúng đắn của thuật toán

Khi một thuật toán đã được chỉ định, bạn phải chứng minh tính đúng đắn của nó. Đó là, bạn phải chứng minh rằng thuật toán mang lại kết quả bắt buộc cho mọi đầu vào hợp pháp trong một khoảng thời gian hữu hạn.

Đối với một số thuật toán, một bằng chứng về tính đúng đắn là khá dễ dàng, đối với những người khác, nó có thể khá phức tạp. Một kỹ thuật phổ biến để chứng minh tính đúng đắn là sử dụng quy nạp toán học vì các phép lặp của thuật toán cung cấp một chuỗi các bước tự nhiên cần thiết cho các chứng minh đó. Điều đáng nói là mặc dù việc theo dõi hiệu suất của thuật toán cho một số đầu vào cụ thể có thể là một hoạt động rất đáng giá, nhưng nó không thể chứng minh tính đúng đắn của thuật toán một cách rõ ràng. Nhưng để chỉ ra rằng một thuật toán không chính xác, bạn chỉ cần một trường hợp đầu vào của nó mà thuật toán không thành công.

Khái niệm về tính đúng cho các thuật toán xấp xỉ ít đơn giản hơn so với các thuật toán chính xác. Đối với một thuật toán gần đúng, chúng tôi thường muốn có thể chỉ ra rằng lỗi do thuật toán tạo ra không vượt quá một giới hạn được xác định trước.

1.2.8 Phân tích một thuật toán

Trên thực tế, có hai loại hiệu quả của thuật toán: hiệu quả về thời gian, cho biết tốc độ chạy của thuật toán và hiệu quả về không gian, cho biết nó sử dụng thêm bao nhiêu bộ nhớ.

Một đặc tính mong muốn khác của một thuật toán là tính đơn giản. Không giống như hiệu quả, có thể được xác định chính xác và nghiên cứu bằng sự nghiêm ngặt của toán học, tính đơn giản, giống như vẻ đẹp, ở một mức độ đáng kể trong mắt người xem.

Tuy nhiên, một đặc tính mong muốn khác của một thuật toán là tính tổng quát. Trên thực tế, có hai vấn đề ở đây: tính tổng quát của vấn đề mà thuật toán giải quyết và tập hợp các đầu vào mà nó chấp nhận. Về vấn đề đầu tiên, hãy lưu ý rằng đôi khi việc thiết kế một thuật toán cho một vấn đề được đặt ra theo các thuật ngữ chung hơn lại dễ dàng hơn.

1.3 Important Problem Types

Trong vô số các vấn đề mà người ta gặp phải trong máy tính, có một số lĩnh vực thu hút sự chú ý đặc biệt của các nhà nghiên cứu. Nhìn chung, sự quan tâm của họ đã được thúc đẩy bởi tầm quan trọng thực tế của vấn đề hoặc bởi một số đặc điểm cụ thể khiến vấn đề trở thành một chủ đề nghiên cứu thú vị; may mắn thay, hai động lực thúc đẩy này củng cố lẫn nhau trong hầu hết các trường hợp.

Trong phần này, chúng tôi sẽ giới thiệu các dạng vấn đề quan trọng nhất:

+ Sorting

+ Searching

+ String processing

+ Graph problems

+ Combinatorial problems

+ Geometric problems

+ Numerical problems

1.3.1 Sorting

Sắp xếp là sắp xếp lại các mục của một danh sách nhất định theo thứ tự không giảm. Tất nhiên, để vấn đề này có ý nghĩa, bản chất của các mục danh sách phải cho phép một thứ tự như vậy. Như một vấn đề thực tế, chúng ta thường cần sắp xếp danh sách các số, các ký tự từ bảng chữ cái, các chuỗi ký tự và quan trọng nhất.

1.3.2 Searching

Bài toán tìm kiếm đề cập đến việc tìm một giá trị nhất định, được gọi là khóa tìm kiếm, trong một tập hợp nhất định (hoặc tập hợp nhiều, cho phép một số phần tử có cùng giá trị). Có rất nhiều thuật toán tìm kiếm để lựa chọn. Chúng bao gồm từ tìm kiếm tuần tự đơn giản đến tìm kiếm nhị phân hiệu quả ngoạn mục nhưng có giới hạn và các thuật toán dựa trên việc biểu diễn tập hợp cơ bản ở một dạng khác có lợi hơn cho việc tìm kiếm.

1.3.3 String Processing

Chuỗi là một chuỗi các ký tự từ một bảng chữ cái.Các chuỗi được quan tâm đặc biệt là các chuỗi văn bản, bao gồm các chữ cái, số và các ký tự đặc biệt; chuỗi bit, bao gồm số không và số một; và trình tự gen, có thể được mô hình hóa bởi các chuỗi ký tự từ bảng chữ cái bốn ký tự {A, C, G, T}. Tuy nhiên, cần phải chỉ ra rằng các thuật toán xử lý chuỗi có quan trọng đối với khoa học máy tính trong một thời gian dài cùng với máy tính ngôn ngữ và các vấn đề biên dịch.

1.3.4 Graph Problems

Một trong những lĩnh vực lâu đời nhất và thú vị nhất trong thuật toán là thuật toán đồ thị. Một cách không chính thức, đồ thị có thể được coi như một tập hợp các điểm được gọi là đỉnh, một số điểm được nối với nhau bằng các đoạn thẳng được gọi là các cạnh. Đồ thị là một đối tượng thú vị để nghiên cứu, vì lý do cả lý thuyết và thực tiễn. Đồ thị có thể được sử dụng để lập mô hình cho nhiều ứng dụng, bao gồm giao thông vận tải, truyền thông, mạng xã hội và kinh tế, lập kế hoạch dự án và trò chơi. Nghiên cứu các khía cạnh kỹ thuật và xã hội khác nhau của Internet nói riêng là một trong những lĩnh vực nghiên cứu tích cực hiện nay liên quan đến các nhà khoa học máy tính, nhà kinh tế học và nhà khoa học xã hội .

Các thuật toán đồ thị cơ bản bao gồm Graph Travelsal Algorithms , Shortest Path Algorithms , Topological Sorting For Graphs With Directed Edges.

1.3.5 Geometric Problems

Các thuật toán hình học xử lý các đối tượng hình học như điểm, đường thẳng và đa giác. Các thuật toán chỉ cho hai vấn đề kinh điển của tính toán hình học: bài toán cặp gần nhất và bài toán lồi cầu.

1.3.6 Numerical Problems

Các vấn đề số, một lĩnh vực ứng dụng đặc biệt lớn khác, là những vấn đề liên quan đến các đối tượng toán học có tính chất liên tục: giải phương trình và hệ thống phương trình, tính toán tích phân xác định, đánh giá các hàm, v.v. Phần lớn các vấn đề toán học như vậy chỉ có thể được giải quyết một cách gần đúng. Một khó khăn chính khác bắt nguồn từ thực tế là những vấn đề như vậy thường yêu cầu thao tác các số thực, chỉ có thể được biểu diễn trong máy tính xấp xỉ. Hơn nữa, một số lượng lớn các phép toán số học được thực hiện trên các con số được đại diện gần đúng có thể dẫn đến sự tích lũy của sai số làm tròn đến một điểm mà nó có thể làm sai lệch đáng kể đầu ra được tạo ra bởi một thuật toán âm thanh.

1.4 Cấu trúc dữ liệu cơ bản

Vì phần lớn các thuật toán quan tâm hoạt động dựa trên dữ liệu, các cách tổ chức dữ liệu cụ thể đóng một vai trò quan trọng trong việc thiết kế và phân tích các thuật toán. Cấu trúc dữ liệu có thể được định nghĩa là một sơ đồ cụ thể để tổ chức các mục dữ liệu liên quan. Bản chất của các mục dữ liệu được quyết định bởi vấn đề hiện tại; họ có thể phạm vi từ các kiểu dữ liệu cơ bản đến cấu trúc dữ liệu. Có một số cấu trúc dữ liệu đã được chứng minh là đặc biệt quan trọng đối với các thuật toán máy tính.

1.4.1 Cấu trúc dữ liệu tuyến tính

Hai cấu trúc dữ liệu cơ bản quan trọng nhất là mảng và danh sách liên kết. Mảng (một chiều) là một chuỗi gồm n mục có cùng kiểu dữ liệu được lưu trữ liền kề trong bộ nhớ máy tính và có thể truy cập được bằng cách chỉ định một giá trị của chỉ mục của mảng

Mảng được sử dụng để triển khai nhiều cấu trúc dữ liệu khác nhau. Nổi bật trong số đó là chuỗi, một chuỗi các ký tự từ bảng chữ cái được kết thúc bằng một ký tự đặc biệt cho biết kết thúc của chuỗi. Các chuỗi bao gồm các số không và một được gọi là chuỗi nhị phân hoặc chuỗi bit. Chuỗi là không thể thiếu để xử lý dữ liệu dạng văn bản, xác định ngôn ngữ máy tính và biên dịch các chương trình được viết trong đó và nghiên cứu các mô hình tính toán trừu tượng. Các hoạt động chúng ta thường thực hiện trên chuỗi khác với các hoạt động chúng ta thường thực hiện trên các mảng khác (ví dụ, mảng số). Chúng bao gồm tính toán độ dài chuỗi, so sánh hai chuỗi để xác định chuỗi nào đứng trước chuỗi kia theo thứ tự từ vựng (tức là theo thứ tự bảng chữ cái) và nối hai chuỗi (tạo thành một chuỗi từ hai chuỗi đã cho bằng cách nối chuỗi thứ hai vào cuối chuỗi thứ nhất).

Danh sách liên kết là một chuỗi gồm không hoặc nhiều phần tử được gọi là các nút, mỗi phần tử chứa hai loại thông tin: một số dữ liệu và một hoặc nhiều liên kết được gọi là con trỏ đến các nút khác của danh sách liên kết. (Một con trỏ đặc biệt được gọi là "null" được sử dụng để chỉ ra sự vắng mặt của nút kế nhiệm.) Trong danh sách được liên kết đơn lẻ, mỗi nút ngoại trừ nút cuối cùng chứa một con trỏ đến phần tử tiếp theo

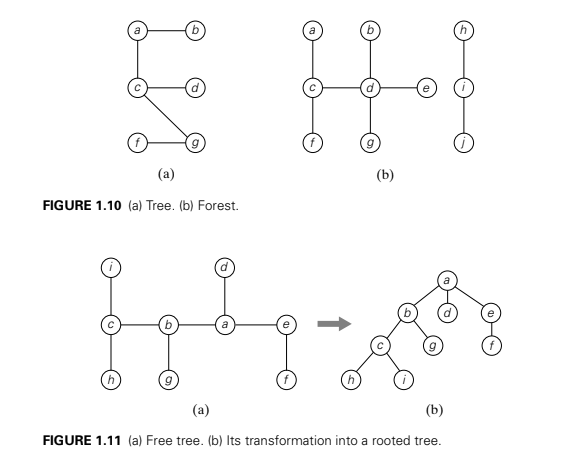
1.4.2 Đồ thị

Như chúng ta đã đề cập trong phần trước, một biểu đồ được coi là một tập hợp các điểm trong mặt phẳng được gọi là “đỉnh” hoặc “nút”, một số trong số chúng được nối với nhau bằng các đoạn thẳng được gọi là “cạnh” hoặc “cung”. Về mặt hình thức, một đồ thị G = V, E được xác định bởi một cặp hai tập hợp: tập hợp hữu hạn vô hạn V gồm các phần được gọi là đỉnh và tập E gồm các cặp phần này được gọi là các cạnh. Nếu các cặp đỉnh này không có thứ tự, tức là một cặp đỉnh (u, v) giống với cặp (v, u), chúng ta nói rằng các đỉnh u và v kề nhau và chúng được nối với nhau bởi cạnh vô hướng (u, v). Chúng ta gọi các đỉnh u và v là điểm cuối của cạnh (u, v) và nói rằng u và v là sự cố đối với cạnh này; chúng tôi cũng nói rằng cạnh (u, v) là sự cố tới các điểm cuối u và v của nó. Một đồ thị G được gọi là vô hướng nếu mọi cạnh trong nó là vô hướng.

1.4.3 Trees

Cây (chính xác hơn, cây tự do) là một đồ thị mạch hở liên thông (Hình 1.10a).

Một đồ thị không có chu trình nhưng không nhất thiết phải liên thông được gọi là rừng: mỗi thành phần liên thông của nó là một cây (Hình 1.10b).



CHƯƠNG 2- BRUTE - FORCE

2.1 Khái niệm:

- Brute Force với ý tưởng sử dụng “Force” với cách tiếp cận vấn để theo kiểu “Brute”.

- Những thuật toán dạng Brute Force tuy không thực sự “Thông minh” và hiệu quả về mặt hiệu suất tuy nhiên chúng có thể áp dụng được cho rất nhiều vấn đề.

- Đối với một số vấn đề như sort, matching .., Brute Force lại là cách tiếp cận hợp lý.

- Đối với các vấn đề nhỏ, Brute Force vẫn thực sự hữu ích để giải quyết.

2.2 Selection Sort và Bubble Sort:

- Nếu hỏi một sinh viên về thuật toán sắp xếp nào hiệu quả nhất thì có thể họ sẽ không có câu trả lời, tuy nhiên nếu hỏi sinh viên sử dụng thuật toán nào thì chắc chắn sẽ là 2 thuật toán dưới đây.

- Trong phần này, chúng ta sẽ thử tiếp cần Brute Force thông qua 2 thuật toán sắp xếp Select Sort và Bubble sort.

2.2.1 Selection Sort:

- Thuật toán với ý tưởng sẽ tìm phần tử nhỏ nhất từ phần tử thứ 0 tới n-1 và hoán đổi với phần tử thứ 0, sau đó tiếp tục tìm phần tử nhỏ nhất cho vị trí tiếp theo( vị trí thứ 1) từ vị trí đó tới phần tử cuối cùng và lặp lại cho tới khi tìm được phần tử nhỏ nhất cho vị trí thứ n-2.

- Mã giả:

1. selectionSort(A[0,…n-1]):
2. // Purpose: Sắp xếp mảng theo thứ tự tăng dần.
3. // Input: Một mảng có kích thước n phần tử
4. //Output: Mảng đầu vào đã được sắp xếp theo thứ tự tăng dần
5. For I = 0 to n-2:
6. Min =i
7. For j = i+1 to n-1:
8. If ( A[j]<A[min]):
9. min = j
10. Endif
11. Endfor
12. Swap (A[min],A[i])
13. Endfor
14. Endalgorithm

* Code hiện thực thuật toán:

# Thuật toán sắp xếp brute force

def selectionSort(A):

    # Purpose: Sắp xếp mảng theo thứ tự tăng dần

    # Input: Một mảng có kích thước n phần tử

    # Output: Mảng đầu vào đã được sắp xếp theo thứ tự tăng dần

    for i in range(0, len(A)-1):

        min = i

        for j in range(i+1, len(A)):

            if A[j] < A[min]:

                min = j

        A[min], A[i] = A[i], A[min]

* Phép toán cơ bản nhất:
  + Phép so sánh tại vị trí 8 A[j]<A[min]
* Độ phức tạp thuật toán:
  + T(n)=1.(n-1 + n-2 + n-3 + …. + 1) =
  + T(n) O()
* Code để test running time với 100 input sizes khác nhau:

def timmerSelectSort():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        start = time.time()

        main.selectionSort(temp)

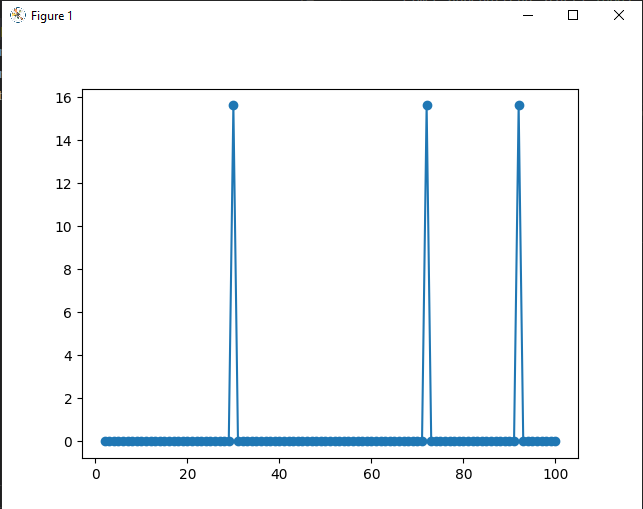
        end = time.time()

        timer.append((end- start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả



2.2.2 Bubble Sort:

- Khác với thuật toán Selection Sort, Bubble Sort được xây dựng bằng cách thực hiện “chuyền” lần lượt phần tử có giá trị lớn nhất từ vị trí đầu tiên cho tới ví trí cuối cùng của lần lặp. Thuật toán sẽ thực hiện so sánh lần lượt 2 cặp số liên tiếp j và j+1.

- Mã giả thuật toán:

1. selectionSort(A[0,…n-1]):
2. // Purpose: Sắp xếp mảng theo thứ tự tăng dần.
3. // Input: Một mảng có kích thước n phần tử
4. //Output: Mảng đầu vào đã được sắp xếp theo thứ tự tăng dần
5. For i= 0 to n-2 :
6. For j=0 to n-2-i:
7. If( A[j+1] <A[j]) :
8. Swap(A[j+1],A[j])
9. Endfor
10. Endfor
11. EndAlgorithm

* Code hiện thực với python:

# Thuật toán bubble sort

def bubbleSort(A):

    # Purpose: Sắp xếp mảng theo thứ tự tăng dần.

    # Input: Một mảng có kích thước n phần tử

    # Output: Mảng đầu vào đã được sắp xếp theo thứ tự tăng dần

    for i in range(len(A)-1):

        for j in range(len(A)-1-i):

            if A[j+1] < A[j]:

                A[j+1], A[j] = A[j], A[j+1]

* Phép toán cơ bản nhất:
  + Phép so sánh tại dòng 7 A[j+1]<A[j]
* Độ phức tạp thuật toán:
  + T(n)=1.(n-1 + n-2 + n-3 + …. + 1) =
  + T(n) O()

- Code Test running time:

def timmerBubbleSort():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        start = time.time()

        main.bubbleSort(temp)

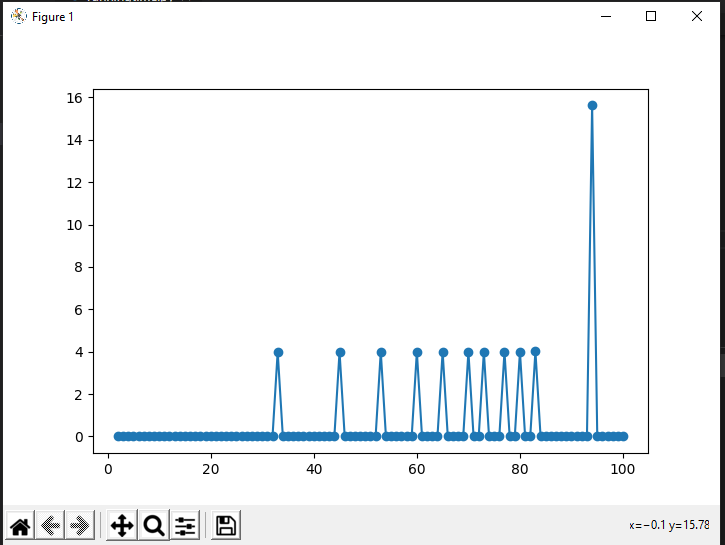
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

- Kết quả:



2.3 Search:

- Phần trước chúng ta đã phân tích một số thuật toán Sort theo chiến lược Brute Force, tiếp theo chúng ta sẽ tìm hiểu một thuật toán tìm kiếm cũng rất phổ biến đó là Sequential Search.

2.3.1 Sequential Search.

- Thuật toán tìm kiếm tuần tự với ý tưởng như tên gọi của nó, tìm kiếm lần lượt từng phần tử để tìm ra phần tử key. Tuy nhiên, hôm nay chúng ta sẽ thực hiện một bản nâng cấp của thuật toán tìm kiếm tuần tự.

- Thuật toán tìm kiếm tuần tự 2 sẽ thêm khóa key vào cuối mảng cần tìm, điều này giúp chúng ta không cần phải kiểm tra khi nào duyệt hết phần tử.

- Mã giả thuật toán:

1. sequentialSearch(A[0,…n],k):
2. // Purpose:Tìm kiếm key k trong mảng một cách tuần tự.
3. // Input: Một mảng cần duyệt có kích thước n phần tử A[0,n-1] và giá trị cần tìm k
4. //Output: index của phần tử cần tìm trong A hoặc -1 nếu không tìm thấy.
5. A[n]=k
6. i=0
7. While A[i]≠k do:
8. i=i+1
9. endwhile
10. If I <n return i
11. Return -1

* Code hiện thực thuật toán với python 3:

def sequentialSearch(A, k):

    # Purpose:Tìm kiếm key k trong mảng một cách tuần tự

    # Input: Một mảng cần duyệt có kích thước n phần tử A[0,n-1] và giá trị cần tìm k

    # Output: index của phần tử cần tìm trong A hoặc -1 nếu không tìm thấy

    A.append(k)

    i = 0

    while A[i] != k:

        i = i+1

    if i < len(A)-1:

        return i

    return -1

* Phép toán cơ bản nhất:
  + Phép so sánh tại dòng 7 A[i]≠k
* Độ phức tạp thuật toán:
  + T(n)=1.n = n
  + T(n) O(n)
* Code test running time:

def timmerSequentialSearch():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        k = random.randint(0, 100)

        start = time.time()

        print(main.sequentialSearch(temp, k))

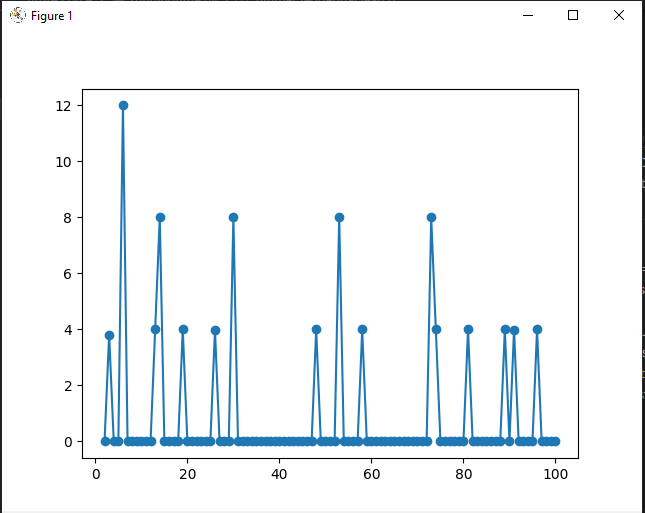
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả:



2.4 Kết luận:

- Sau khi tìm hiểu qua 3 đại diện tiêu biểu về Brute Force. Selection Sort, Bubble Sort và Sequential Search. Chúng ta có thể rút ra một số nhận định sau:

* Brute Force là một cách đơn giản để tiếp cận vấn đề. “Nghĩ sao làm vậy, không chú trọng hiệu năng, chỉ quan tâm kết quả”.
* Điểm mạnh của phương pháp này đó là có thể áp dụng được vào hầu hết các vấn đề, tuy nhiên điều này có thể dẫn nhiều vấn đề trong kĩ thuật lập trình.
  + Ví dụ, Một bài toán với đầu vào là một mảng đã được sắp xếp, không nên sử dụng thuật toán tìm kiếm tuần tự cho điều này. Hay thuật toán Bubble Sort và Selection Sort với độ phức tạp O() do đó hiện nay không ai sử dụng chúng vào các dự án phức tạp.
* Tuy nhiên, nói như vậy không có nghĩa không nên sử dụng Brute Force. Ở một vài khía cạch, chúng ta buộc phải sử dụng tới chiến lược này, ví dụ như thuật toán tìm kiếm đối với môt mảng đầu vào chưa được sắp xếp, chúng ta buộc phải sử dụng thuật toán tìm kiếm tuần tự thay vì tìm kiếm nhị phân, hay trong các vấn đề về bảo mật, các hacker thường xuyên thực hiện chiến lược này để tìm thông tin người dùng, do đó chúng ta cần phải hiểu thật rõ chiến lược này.

CHƯƠNG 3 – DIVIDE AND CONQUER

- Trong phần này chúng ta sẽ tìm hiểu về chiến lược “Chia để trị”, cái tên mà học sinh phổ thông nào cũng được nghe tới thông qua môn lịch sử.

- Divde: chia nhỏ vấn đề và Conquer: giải quyết các vấn đề nhỏ đó.

- Các thuật toán Divide and Conquer hoạt động theo cách sau:

* Một vấn đề được chia làm nhiều bài toán con cùng loại. (Thường là cùng kích thước).
* Thường được thực hiện bởi các thuật toán đệ quy.
* Các giải pháp giải quyết các vấn đề con được kết hợp để giải quyết vấn đề ban đầu.

3.1 Merge Sort

- Merge Sort là một ví dụ hoàn hảo về việc áp dụng thành công kĩ thuật Divide and Conquer để thực hiện sắp xếp.

- Merge Sort thực hiện trên ý tưởng chia mảng hiện tại thành 2 nửa A[0,…n/2 -1] và A[n/2,…n-1] cho đến khi mảng còn 1 phâng tử, và tiến hành sắp xếp chúng theo cách đệ quy. Và cuối cùng hợp nhất mảng con thành một mảng được sắp xếp.

- Mã giả hàm mergesort:

1. Mergesort(A[0,…n-1]):
2. // sắp xếp mảng bởi đệ quy merge sort.
3. //input: 1 mảng A có kích thước n phần tử.
4. //output: Mảng được sắp xếp tăng dần.
5. If(n >1):
6. Copy A[0,..n/2-1] to B[0,…n/2-1]
7. Copy A[n/2, …n-1] to C[0,…n/2-1]
8. Mergesort(B)
9. Mergesort(C)
10. Merge(B,C,A) \*(1)\*
11. Endif
12. EndAlgorithm

* (1) Giai đoạn merge 2 mảng con thành một mảng sorted A.
* Hàm Merge sẽ thực hiện so sanh lần lượt 2 cặp số từ 2 mảng đầu vào, giá trị nhỏ hơn sẽ được add vào phần tử đầu tiên của mảng A. Quá trinh này sẽ được lặp đi lặp lại cho tới khi duyệt hết 1 trong 2 mảng đầu vào. Và tiếp tục copy các giá trị còn lại của mảng chưa được duyệt hết vào mảng A.
* Mã giả hàm merge:

1. Merge(B,C,A):
2. // purpose : Hợp 2 mảng B C thành mảng A theo thứ tự tăng dần.
3. // Input: mảng B có kích thước p, mảng C có kích thước q, mảng A có kích thước p+q
4. // Ouput: Sắp xếp mảng A từ mảng B và C
5. i, j, k=0
6. While i <p and j <q do:
7. If B[i]≤C[j]:
8. A[k]=B[i]; i+=1
9. Else :
10. A[k]=C[j]; j+=1
11. Endif
12. k+=1
13. Endwhile
14. If i =p:
15. Copy C[j,q-1] to A[k,…p+q-1]
16. Else:
17. Copy B[i,p-1] to A[k,…p+q-1]
18. Endif
19. Endalgorithm

* Code hiện thực thuật toán với python:

def mergeSort(A):

    # sắp xếp mảng bởi đệ quy merge sort.

    # input :1 mảng A có kích thước n phần tử

    # output: Mảng được sắp xếp tăng dần

    if len(A) > 1:

        B = []

        C = []

        for i in range(len(A)//2):

            B.append(A[i])

        for i in range(len(A)//2, len(A)):

            C.append(A[i])

        mergeSort(B)

        mergeSort(C)

        merge(B, C, A)

def merge(B, C, A):

    # purpose : Hợp 2 mảng B C thành mảng A theo thứ tự tăng dần.

    # Input: mảng B có kích thước p, mảng C có kích thước q, mảng A có kích thước p+q

    # Ouput: Sắp xếp mảng A từ mảng B và C

    i = 0

    j = 0

    k = 0

    while i < len(B) and j < len(C):

        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 == len(B):

        for n in range(j, len(C)):

            A[k] = C[n]

            k = k+1

    else:

        for n in range(i, len(B)):

            A[k] = B[n]

            k = k+1

* Độ phức tạp của thuật toán:
* T(n) = 2T(n/2) + fmerge (n)
* fmerge (n), chúng ta sẽ xem xét số lần thực hiện phép so sánh tại dòng 7 B[i]≤C[j].
* Phép so sánh sẽ được thực hiện cho tới khi duyệt hết một trong 2 mảng B và C, trong trường hợp tệ nhất, ví dụ mảng B[1,3,5] và mảng C[2,4,6], tổng số lần thực hiện phép so sánh sẽ là 5. Vậy trong trường hợp tệ nhất fmerge (n)= n -1.
* Vậy T(n) = 2T(n/2)+n-1.
* Áp dụng định lý Master Theorem. a = 2, b=2, k=1
  + => T(n) (nlogn)

- Code test running time:

def timmerMergeSort():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        start = time.time()

        main.mergeSort(temp)

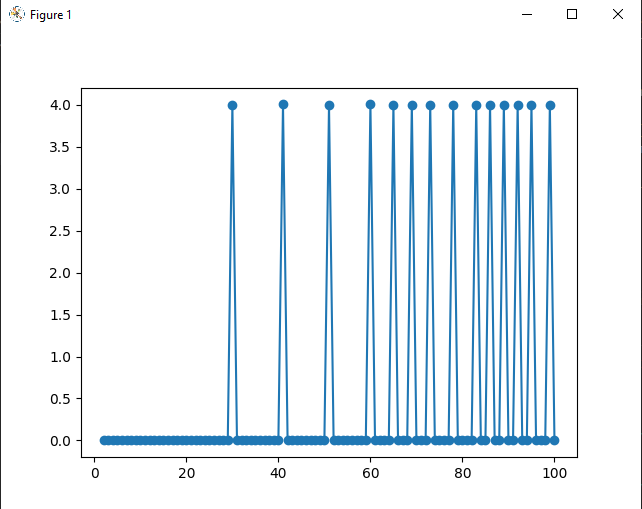
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả:



3.2 Quick Sort

- Tiếp theo chúng ta sẽ xem xét một thuật toán Sort khác, Quick Sort.

- Cũng giống như Merge Sort, thuật toán Quick Sort cũng là một thuật toán chia để trị.

- Ý tưởng của thuật toán Quick Sort là lựa chọn một điểm đánh dấu pivot, thuật toán sẽ thực hiện chia mảng thành các mảng con dựa vào pivot đã chọn. Di chuyển tất cả các giá trị nhỏ hơn pivot sang bên trái và các giá trị lớn hơn qua bên phải so với pivot.

- Mã giả:

1. Quicksort(A,l,r)
2. // mục đích: sắp xếp mảng theo thuật toán quicksort
3. // input: một mảng A và l là index đầu và r là index của phần tử cuối trong mảng.
4. // output: mảng A được sắp xếp tăng dần.
5. If l < r
6. s = partition(A,l,r) \*(1)\*
7. Quicksort(A,l,s-1)
8. Quicksort(A,s+1,r)
9. Endif
10. Endalgorithm

* (1) Hàm partition sẽ thực hiện trả về vị trí của pivot.
* Mã giả:

1. partition(A,l,r):
2. // mục đích, đẩy các giả trị nhỏ hơn pivot sang bên trái và giá trị lớn hơn sang bên phải
3. // input Mảng A và index l của phần tử đầu tiền và index r của phần tử cuối cùng trong Mảng A hiện tại
4. // output Trả về index của pivot hiện tại trong mảng A.
5. p = A[l]
6. i = l
7. j = r
8. while i < j do:
9. i=i+1
10. while A[i]<p do :
11. i++ // tại đây chúng ta cần kiểm tra i ≤r
12. endwhile
13. While A[j]>p do:
14. J -- // tại đây chúng ta cần kiểm tra j< l
15. Endwhile
16. Swap(A[i],A[j])
17. Endwhile
18. Swap(A[i],A[j])
19. Swap(A[l],A[j]) // đổi vị trí pivot ban đầu so với vị trí pivot mới tìm
20. Return j

* Code hiện thực thuật toán với python 3:

def quickSort(A, l, r):

    # mục đích: sắp xếp mảng theo thuật toán quicksort

    # input: một mảng A và l là index đầu và r là index của phần tử cuối trong mảng

    # output: mảng A được sắp xếp tăng dần.

    if l < r:

        s = partition(A, l, r)

        quickSort(A, l, s-1)

        quickSort(A, s+1, r)

def partition(A, l, r):

    # mục đích, đẩy các giả trị nhỏ hơn pivot sang bên trái và giá trị lớn hơn sang bên phải

    # input Mảng A và index l của phần tử đầu tiền và index r của phần tử cuối cùng trong Mảng A hiện tại

    # output Trả về index của pivot hiện tại trong mảng A.

    p = A[l]

    i = l

    j = r

    while i < j:

        i = i+1

        while A[i] < p:

            i = i+1

            if i > r:

                i = r

                break

        while A[j] > p:

            j = j-1

            if j < l:

                j = l

                break

        A[i], A[j] = A[j], A[i]

    A[i], A[j] = A[j], A[i]

    A[l], A[j] = A[j], A[l]

    return j

* Độ phức tạp của thuật toán:
  + T(n) = 2T(n/2)+ fparition(n)
  + fparition(n) = n
  + Trong trường hợp tệ nhất pivot được chọn là giá trị nhỏ nhất hoặc lớn nhất trong mảng A, thì biểu thức T(n) sẽ trở thành T(n)=T(n-1) + n => T(n) O()
  + Trường hợp tốt nhất pivot được chọn là giá trị trung bình (số phần tử bên trái = số phần tử bên phải) thì biểu thức T(n) sẽ trở thành
    - T(n) = 2T(n/2)+n
    - Áp dụng Master Theorem: a= 2, b= 2 k =1
    - => T(n) (nlogn)
* Code test running time:

def timmerQuickSort():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        start = time.time()

        main.quickSort(temp, 0, len(temp)-1)

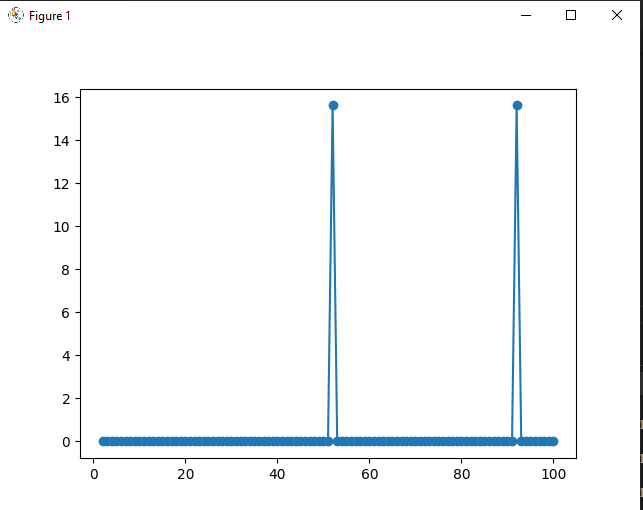
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả:



3.3 Binary Tree Traversals

- Trong phần này chúng ta sẽ xem chiến lược Divide and Conquer đối với cây nhị phân.

- Cây nhị phân được biết đến là cấu trúc dữ liệu dạng cây và có mỗi nút nhiều nhất gồm 2 node con, node trái và phải.

- Chiều cao của cây nhị phân được tính từ node root đến node con xa nhất.

- Thuật toán đệ quy tính chiều cao cây nhị phân T áp dụng Divide and Conquer.

- Mã giả:

1. Height(T):
2. // Tính chiều cao của cây nhị phân
3. // input: Cây nhị phân T
4. // output chiều cao cây T
5. // if T = : return -1
6. // return max( Height(T­trái),Height(Tphải))+1
7. Endalgorithm

* Code:

def height(T):

    # Tính chiều cao của cây nhị phân

    # input: Cây nhị phân T

    # output chiều cao cây T

    if T == None:

        return -1

    return max(height(T.left), height(T.right))+1

class node:

    def \_\_init\_\_(self, left=None, right=None):

        self.left = left

        self.right = right

# Hàm tạo cây binary tree để thuận tiên test running time

def initialBinaryTree(T, n):

    if n == 0:

        return

    T.left = node()

    T.right = node()

    initialBinaryTree(T.left, n-1)

    initialBinaryTree(T.right, n-1)

* Thao tác cơ bản nhất:
  + So sánh tại dòng 5 T =
* Hệ thức truy hồi:
  + A(Tn)= A(Ttrái)+A(Tphải) + 1 ; n là tổng số node
* Độ phức tạp của thuật toán:
  + A(Tn) = n
* Code running time:

def timmerHeight():

    sizeInput = []

    timer = []

    for i in range(0, 20):

        sizeInput.append(i)

        temp = main.node()

        main.initialBinaryTree(temp, i)  # Hàm này tốn rất nhiều thời gian

        start = time.time()

        main.height(temp)

        end = time.time()

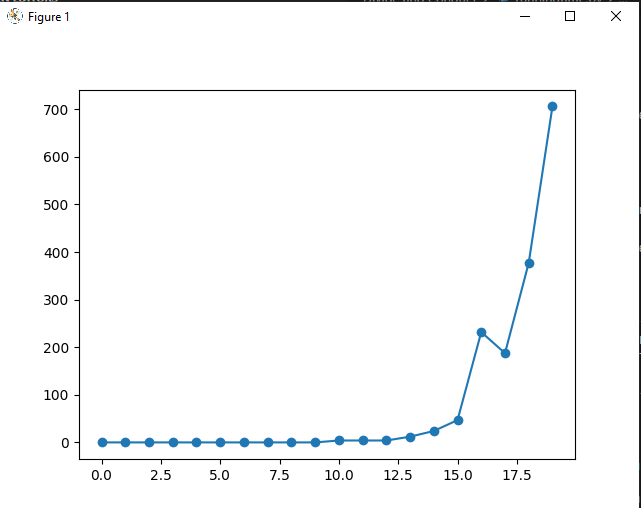
        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    print(timer)

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả:



3.4 Kết luận:

- Sau khi đi qua 3 thuật toán Merge Sort, Quick Sort và Binary Tree Traversals chúng ta có thể thấy rằng :

* Divide and Conquer là một kỹ thuật thiết kế thuật toán chung bằng cách chia nhỏ bài toán thành các bài toán nhỏ hơn cùng loại, và giải quyết từng bài toán nhỏ bằng đệ quy và kết hợp chúng để giải quyết vấn đề ban đầu.
* Thời gian chạy của các thuật toán chia để trị hoàn toàn tối ưu hơn so với kĩ thuật Brute Force, cụ thể với quick sort hay merge sort có độ phức tạp nhỏ hơn nhiều so với 2 thuật toán sort sử dụng kĩ thuật brute force. Tuy nhiên, với thuật toán merge sort lại đến từ việc hao tốn bộ nhớ khi thực hiện khởi tạo quá nhiều mảng con.
* Do vậy, không phải mọi thuật toán Divide and Conquer đều hiệu quả hơn Brute Force, trên thực tế cách tiếp cận chia để trị mang lại một số thuật toán quan trọng và hiệu quả nhất trong khoa học máy tính, trong đó khi các vấn đề con được giải quyết đồng thời.

CHƯƠNG 4- GREEDY ALGORITHMS

- Thuật toán Greedy là một thuật toán đơn giản, trực quan được sử dụng trong các bài toán tối ưu hóa. Thuật toán đưa ra lựa chọn tối ưu ở mỗi bước khi nó cố gắng tìm ra cách tối ưu tổng thể để giải quyết toàn bộ vấn đề. Các thuật toán Greedy khá thành công trong một số vấn đề, chẳng hạn như mã hóa Huffman được sử dụng để nén dữ liệu, hoặc thuật toán Dijkstra, được sử dụng để tìm đường đi ngắn nhất qua biểu đồ.

- Thuật toán Greedy lấy tất cả dữ liệu trong một vấn đề cụ thể, sau đó đặt quy tắc cho các yếu tố nào cần thêm vào giải pháp ở mỗi bước của thuật toán.

- Nếu cả hai thuộc tính dưới đây đều đúng,thuật toán Greedy có thể được sử dụng để giải quyết vấn đề :

* Thuộc tính lựa chọn Greedy : Có thể đạt được giải pháp tối ưu toàn cầu (tổng thể) bằng cách chọn lựa chọn tối ưu ở mỗi bước.
* Cấu trúc con tối ưu: Một bài toán có cấu trúc con tối ưu nếu một giải pháp tối ưu cho toàn bộ bài toán chứa các giải pháp tối ưu cho các bài toán con.

- Nói cách khác, các thuật toán Greedy hoạt động trên các bài toán mà đúng là ở mỗi bước, có một lựa chọn tối ưu cho bài toán cho đến bước đó và sau bước cuối cùng, thuật toán tạo ra giải pháp tối ưu hoàn chỉnh vấn đề.

- Để thực hiện một thuật toán Greedy, hãy xác định một cấu trúc con hoặc bài toán con tối ưu trong bài toán. Sau đó, xác định giải pháp sẽ bao gồm những gì (ví dụ: tổng lớn nhất, đường đi ngắn nhất, v.v.). Tạo một số cách lặp đi lặp lại để đi qua tất cả các bài toán con và xây dựng giải pháp.

- Sau đây, chúng ta sẽ tìm hiểu về 3 thuật toán liên quan tới đồ thị mà đã học ở môn Toán tổ hợp và đồ thị đó là Prim, Kruskal và Dijkstra.

4.1 Tìm cây khung nhỏ nhất

- Thuật toán Prim và Kruskal là hai thuật toán trong lý thuyết đồ thị để tìm cây khung nhỏ nhất của một đồ thị liên thông.

- Định nghĩa: Cây khung hay cây bao trùm là một đồ thị con có dạng cây và có tất cả các đỉnh liên thông với nhau, một đồ thị có rất nhiều cây khung do đó cây khung nhỏ nhất là cây có trọng số bé nhất.

4.1.1 Thuật toán Prim

- Ý tưởng: Thuật toán sẽ bắt đầu duyệt từ một đỉnh và nạp dần các đỉnh còn lại vào cây khung sao cho ở mỗi lần nạp, đỉnh được nạp kề và gần nhất đối với các đỉnh đã được nạp.

- Mã giả :

1. Thuật toán Prim tìm cây bao trùm tối thiểu
2. //Input : Đồ thị liên thông có trọng số G = V,E
3. //Output : ET  tập hợp các cạnh tạo nên một cây khung tối thiểu của G
4. VT 🡨 {v0} // tập hợp các đỉnh cây có thể được khởi tạo với bất kì đỉnh nào
5. ET 🡨 ∅
6. for i 🡨 1 to |V| - 1 do
7. //Tìm cạnh tối thiểu e\* = (v\*,u\*) trong tất cả các cạnh (v,u) sao cho v trong VT //và u trong V – VT
8. VT 🡨 VT ∪ {u\*}
9. ET 🡨 ET ∪ {e\*}
10. Endfor
11. return ET

* Code hiện thực thuật toán với python 3:

from collections import defaultdict

def prim(graph):

    key = [float("inf")] \* len(graph[0])

    parent = [None] \* len(graph[0])

    key[0] = 0

    mstSet = [False] \* len(graph[0])

    parent[0] = -1

    for i in range(len(graph[0])):

        u = minKey(key, mstSet, len(graph[0]))

        mstSet[u] = True

        for v in range(len(graph[0])):

            if graph[u][v] > 0 and mstSet[v] == False and key[v] > graph[u][v]:

                key[v] = graph[u][v]

                parent[v] = u

    result = []

    for i in range(1, len(graph[0])):

        result.append(repr(parent[i]) + "-" + repr(i))

    return result

def minKey(key, mstSet, v):

    min = float("inf")

    for v in range(v):

        if key[v] < min and mstSet[v] == False:

            min = key[v]

            min\_index = v

    return min\_index

* Code test running time thuật toán prim và test cases:

testcase = [[0, 9, 0, 4, 0, 2, 7, 1, 8, 2, 0, 0, 4, 3, 8, 8, 4, 5, 8, 1], [

            9, 0, 9, 1, 5, 7, 6, 5, 8, 4, 3, 1, 7, 0, 2, 0, 7, 8, 8, 5], [

            0, 9, 0, 0, 8, 7, 4, 6, 6, 8, 7, 0, 0, 9, 3, 5, 5, 6, 3, 0], [

            4, 1, 0, 0, 0, 7, 8, 6, 8, 5, 0, 5, 7, 3, 8, 7, 5, 6, 5, 1], [

            0, 5, 8, 0, 0, 5, 6, 4, 9, 9, 5, 2, 0, 2, 1, 5, 4, 0, 7, 7], [

            2, 7, 7, 7, 5, 0, 4, 6, 1, 6, 2, 6, 3, 1, 0, 2, 7, 9, 9, 6], [

            7, 6, 4, 8, 6, 4, 0, 1, 8, 8, 9, 3, 0, 3, 9, 8, 7, 1, 2, 9], [

            1, 5, 6, 6, 4, 6, 1, 0, 8, 2, 0, 7, 2, 5, 5, 7, 8, 4, 0, 4], [

            8, 8, 6, 8, 9, 1, 8, 8, 0, 9, 2, 6, 7, 4, 5, 0, 2, 3, 3, 9], [

            2, 4, 8, 5, 9, 6, 8, 2, 9, 0, 4, 2, 1, 6, 3, 6, 6, 8, 1, 5], [

            0, 3, 7, 0, 5, 2, 9, 0, 2, 4, 0, 1, 6, 1, 0, 6, 8, 8, 2, 6], [

            0, 1, 0, 5, 2, 6, 3, 7, 6, 2, 1, 0, 4, 9, 0, 2, 3, 4, 2, 0], [

            4, 7, 0, 7, 0, 3, 0, 2, 7, 1, 6, 4, 0, 3, 2, 5, 6, 3, 1, 2], [

            3, 0, 9, 3, 2, 1, 3, 5, 4, 6, 1, 9, 3, 0, 8, 0, 2, 9, 7, 5], [

            8, 2, 3, 8, 1, 0, 9, 5, 5, 3, 0, 0, 2, 8, 0, 7, 3, 3, 7, 4], [

            8, 0, 5, 7, 5, 2, 8, 7, 0, 6, 6, 2, 5, 0, 7, 0, 8, 2, 5, 8], [

            4, 7, 5, 5, 4, 7, 7, 8, 2, 6, 8, 3, 6, 2, 3, 8, 0, 0, 1, 1], [

            5, 8, 6, 6, 0, 9, 1, 4, 3, 8, 8, 4, 3, 9, 3, 2, 0, 0, 3, 0], [

            8, 8, 3, 5, 7, 9, 2, 0, 3, 1, 2, 2, 1, 7, 7, 5, 1, 3, 0, 4], [

            1, 5, 0, 1, 7, 6, 9, 4, 9, 5, 6, 0, 2, 5, 4, 8, 1, 0, 4, 0]

            ]

Code test running time:

def timmerPrim():

    sizeInput = []

    timer = []

    for i in range(2, 21):

        sizeInput.append(i)

        temp = [[testcase[n][m] for m in range(i)] for n in range(i)]

        start = time.time()

        main.prim(temp)

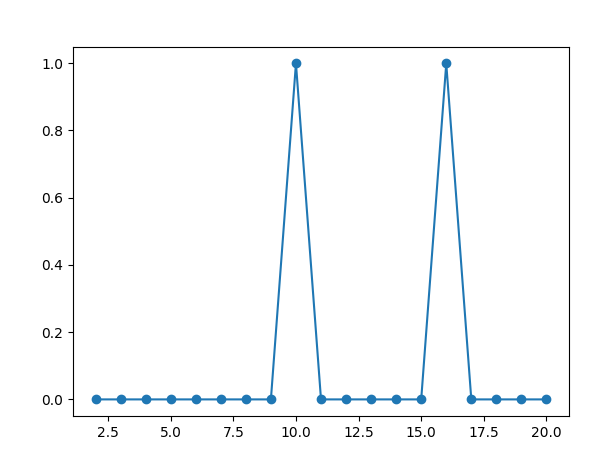
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

Kết quả :



4.1.2 Kruskal’s Algorithm

Trong phần trước, chúng ta đã xem xét thuật toán tham lam “phát triển” một cây bao trùm tối thiểu thông qua việc đưa vào tham lam của đỉnh gần nhất với các đỉnh đã có trong cây. Đáng chú ý, có một thuật toán tham lam khác cho bài toán cây bao trùm tối thiểu cũng luôn đưa ra giải pháp tối ưu. Nó được đặt tên là thuật toán Kruskal theo tên của Joseph Kruskal, người đã khám phá ra thuật toán này khi anh ấy là sinh viên năm thứ hai tốt nghiệp [Kru56]. Thuật toán của Kruskal xem xét cây bao trùm tối thiểu của một đồ thị liên thông có trọng số G = V, E dưới dạng một đồ thị con xoay chiều với |V| - 1 cạnh mà tổng trọng số các cạnh là nhỏ nhất. (Không khó để chứng minh rằng một đồ thị con như vậy phải là một cái cây.) Do đó,thuật toán xây dựng một cây bao trùm tối thiểu dưới dạng một chuỗi mở rộng của các đồ thị con luôn có tính tuần hoàn nhưng không nhất thiết phải được kết nối trên các giai đoạn trung gian của thuật toán.

Thuật toán bắt đầu bằng cách sắp xếp các cạnh của biểu đồ theo thứ tự không giảm về trọng số của chúng. Sau đó, bắt đầu với đồ thị con trống, nó sẽ quét danh sách đã sắp xếp này, thêm cạnh tiếp theo trên danh sách vào đồ thị con hiện tại nếu việc bao gồm như vậy không không tạo một chu trình và chỉ cần bỏ qua cạnh khác.

- Mã giả :

//Thuật toán của Kruskal để xây dựng cây bao trùm tối thiểu

2. //Input : Đồ thị được kết nối có trọng số G = {V,E}

3.//Output: ET, tập hợp các cạnh tạo cây khung tối thiểu G sắp xếp E theo thứ tự không giảm của trọng số cạnh w(ei1)≤… ≤w(ei|E|)

4.while ecounter < |V| - 1 do

5.k <- k + 1

6. if ET ∪ {eik} là mạch vòng

7. ET <- ET ∪ {eik}; ecounter <- ecounter + 1

8. return ET

* Code hiện thực thuật toán:

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = []

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    def find(self, parent, i):

        if parent[i] == i:

            return i

        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):

        xroot = self.find(parent, x)

        yroot = self.find(parent, y)

        if rank[xroot] < rank[yroot]:

            parent[xroot] = yroot

        elif rank[xroot] > rank[yroot]:

            parent[yroot] = xroot

        else:

            parent[yroot] = xroot

            rank[xroot] += 1

def kruskal(Graph):

    result = []

    i = 0

    e = 0

    Graph.graph = sorted(Graph.graph, key=lambda item: item[2])

    parent = []

    rank = []

    for node in range(Graph.V):

        parent.append(node)

        rank.append(0)

    while e < Graph.V - 1:

        u, v, w = Graph.graph[i]

        i = i + 1

        x = Graph.find(parent, u)

        y = Graph.find(parent, v)

        if x != y:

            e = e + 1

            result.append([u, v, w])

            Graph.union(parent, rank, x, y)

    minimumCost = 0

    minPath = []

    for u, v, weight in result:

        minimumCost += weight

        minPath.append(repr(u)+"-"+repr(v))

    return minPath

* Code test running time:
  + Test cases:

g\_2 = main.Graph(10)

g\_2.addEdge(0, 1, 10)

g\_2.addEdge(0, 2, 6)

g\_2.addEdge(0, 3, 5)

g\_2.addEdge(0, 4, 8)

g\_2.addEdge(0, 5, 9)

g\_2.addEdge(0, 6, 12)

g\_2.addEdge(0, 7, 14)

g\_2.addEdge(0, 8, 4)

g\_2.addEdge(0, 9, 2)

g\_2.addEdge(1, 9, 1)

g\_2.addEdge(2, 3, 4)

g\_1 = main.Graph(4)

g\_1.addEdge(0, 1, 10)

g\_1.addEdge(0, 2, 6)

g\_1.addEdge(0, 3, 5)

g\_1.addEdge(1, 3, 15)

g\_1.addEdge(2, 3, 4)

g\_3 = main.Graph(15)

g\_3.addEdge(0, 1, 10)

g\_3.addEdge(0, 2, 6)

g\_3.addEdge(0, 3, 5)

g\_3.addEdge(0, 4, 8)

g\_3.addEdge(0, 5, 9)

g\_3.addEdge(0, 6, 12)

g\_3.addEdge(0, 7, 14)

g\_3.addEdge(0, 8, 4)

g\_3.addEdge(0, 9, 2)

g\_3.addEdge(0, 10, 1)

g\_3.addEdge(0, 11, 8)

g\_3.addEdge(0, 12, 7)

g\_3.addEdge(0, 13, 14)

g\_3.addEdge(1, 14, 20)

g\_3.addEdge(2, 3, 4)

g\_4 = main.Graph(25)

g\_4.addEdge(0, 1, 10)

g\_4.addEdge(0, 2, 6)

g\_4.addEdge(0, 3, 5)

g\_4.addEdge(0, 4, 8)

g\_4.addEdge(0, 5, 9)

g\_4.addEdge(0, 6, 12)

g\_4.addEdge(0, 7, 14)

g\_4.addEdge(0, 8, 4)

g\_4.addEdge(0, 9, 2)

g\_4.addEdge(0, 10, 1)

g\_4.addEdge(0, 11, 8)

g\_4.addEdge(0, 12, 7)

g\_4.addEdge(0, 13, 14)

g\_4.addEdge(1, 14, 28)

g\_4.addEdge(1, 15, 22)

g\_4.addEdge(1, 16, 21)

g\_4.addEdge(1, 17, 20)

g\_4.addEdge(1, 18, 15)

g\_4.addEdge(1, 19, 13)

g\_4.addEdge(2, 3, 4)

g\_4.addEdge(2, 20, 77)

g\_4.addEdge(2, 21, 8)

g\_4.addEdge(2, 22, 9)

g\_4.addEdge(2, 23, 3)

g\_4.addEdge(23, 24, 2)

- Code test:

mycase = [g\_1, g\_2, g\_3, g\_4]

def timmerKruskal():

    sizeInput = []

    timer = []

    for i in range(len(mycase)):

        sizeInput.append(mycase[i].V)

        start = time.time()

        print(main.kruskal(mycase[i]))

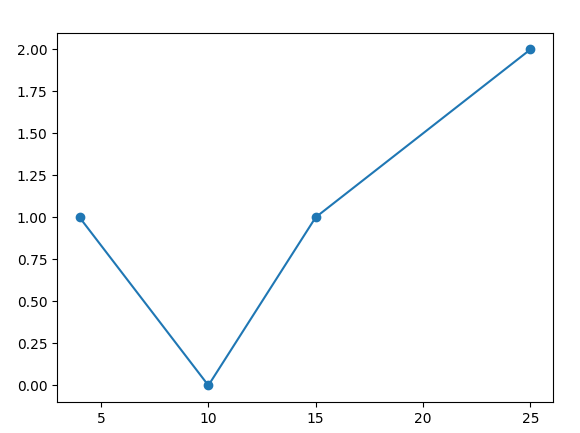
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

- Kết quả:



4.2 Dijkstra’s Algorithm

Thuật toán Dijkstra là một thuật toán để tìm các đường đi ngắn nhất giữa các nút trong biểu đồ, có thể biểu diễn. Nó được hình thành bởi nhà khoa học máy tính Edsger W. Dijkstra vào năm 1956 và được xuất bản ba năm sau đó.

Thuật toán tồn tại trong nhiều biến thể. Thuật toán ban đầu của Dijkstra đã tìm ra đường đi ngắn nhất giữa hai nút nhất định, nhưng một biến thể phổ biến hơn sửa một nút duy nhất làm nút "nguồn" và tìm đường đi ngắn nhất từ ​​nguồn đến tất cả các nút khác trong biểu đồ, tạo ra một đường ngắn nhất cây.

Thuật toán Dijkstra sử dụng các nhãn là số nguyên dương hoặc số thực, được sắp xếp hoàn toàn. Nó có thể được tổng quát để sử dụng bất kỳ nhãn nào được sắp xếp một phần, miễn là các nhãn tiếp theo (nhãn tiếp theo được tạo ra khi đi qua một cạnh) là đơn nguyên không giảm. Tổng quát hóa này được gọi là thuật toán đường đi ngắn nhất Dijkstra chung.

- Mã giả thuật toán:

1// Thuật toán Dijkstra cho các đường đi ngắn nhất nguồn đơn

2. // Đầu vào: Đồ thị liên thông có trọng số G = V, E với các trọng số không âm   
và đỉnh của nó là s

3. // Đầu ra: Độ dài dv của đường đi ngắn nhất từ ​​s đến v và đỉnh áp chót của nó pv cho mọi đỉnh v trong V. Khởi tạo (Q) khởi tạo hàng đợi ưu tiên để trống

4. for every vertex v in V

5. dv <- ∞; pv <- null

6. Insert (Q,v,dv) // khởi tạo ưu tiên đỉnh trong hàng đợi ưu tiên ds <- 0;

Giảm (Q,s,ds) // cập nhật mức độ ưu tiên của s với ds VT <- ∅

7.for i <- 0 to |V| - 1 do

8. u\* <- DeleteMin(Q) //xóa phần tử ưu tiên tối thiểu

9. VT <- VT ∪ {u∗}

10. for every vertex u in V – VT that is adjacent to u\* do

11.if du\* + w (u\* , u ) < du

12.du <- du\* + w (u\* , u ) ; Pu <- u\*

13.Giảm xuống(Q,u,du)

* Code hiện thực thuật toán:

# Thuật toán dijkstras

def dijkstra(graph, src):

    dist = [float('inf')] \* len(graph[0])

    dist[src] = 0

    sptSet = [False] \* len(graph[0])

    for count in range(len(graph[0])):

        u = minDistance(len(graph[0]), dist, sptSet)

        sptSet[u] = True

        for v in range(len(graph[0])):

            if graph[u][v] > 0 and \

                    sptSet[v] == False and \

                    dist[v] > dist[u] + graph[u][v]:

                dist[v] = dist[u] + graph[u][v]

    return dist

def minDistance(V, dist, sptSet):

    min = float("inf")

    for v in range(V):

        if dist[v] < min and sptSet[v] == False:

            min = dist[v]

            min\_index = v

    return min\_index

* Test code running time với test case tương tự thuật toán prim:
* Kết quả:

def timmer():

    sizeInput = []

    timer = []

    for i in range(2, 21):

        sizeInput.append(i)

        temp = [[testcase[n][m] for m in range(i)] for n in range(i)]

        start = time.time()

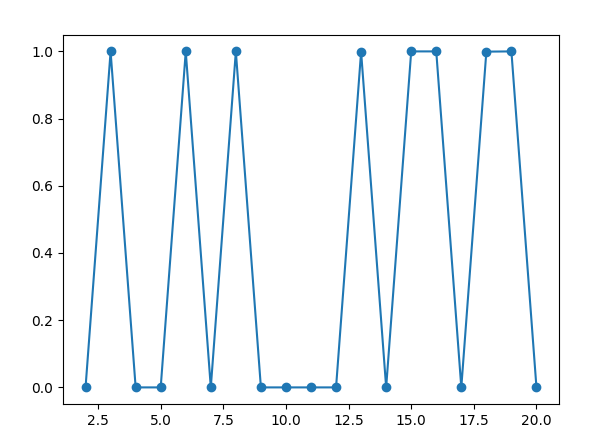
        print(main.dijkstra(temp, 0))

        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()



4.3 Kết luận

- Kỹ thuật Greedy đề xuất xây dựng một giải pháp cho một vấn đề tối ưu hóa thông qua một chuỗi các bước, mỗi bước mở rộng một phần giải pháp thu được cho đến khi đạt được giải pháp hoàn chỉnh cho vấn đề. Trên mỗi bước, lựa chọn được đưa ra phải khả thi, tối ưu cục bộ và không thể thu hồi.

- Thuật toán của Prim là một thuật toán Greedy để xây dựng một khoảng mở rộng tối thiểu cây của một đồ thị liên thông có trọng số. Nó hoạt động bằng cách gắn vào một cây con đã xây dựng một đỉnh gần nhất với các đỉnh đã có trong cây.

- Thuật toán của Kruskal là một thuật toán Greedy khác để kéo dài tối thiểu vấn đề cây. Nó xây dựng một cây bao trùm tối thiểu bằng cách chọn các cạnh theo thứ tự không giảm về trọng lượng của chúng với điều kiện là sự bao gồm không tạo ra một chu kỳ. Kiểm tra điều kiện sau một cách hiệu quả yêu cầu một ứng dụng của một trong những thuật toán được gọi là liên hợp tìm.

- Thuật toán của Dijkstra giải quyết vấn đề tìm đường ngắn nhất nguồn đơn

đường đi ngắn nhất từ ​​một đỉnh đã cho (nguồn) đến tất cả các đỉnh khác của đồ thị hoặc đồ thị có trọng số. Nó hoạt động như thuật toán của Prim nhưng so sánh đường dẫn độ dài chứ không phải độ dài cạnh. Thuật toán của Dijkstra luôn mang lại kết quả chính xác giải pháp cho một đồ thị có trọng số không âm.

CHƯƠNG 5 – DYNAMIC PROGRAMMING

- Sau khi đã tìm hiểu xong kĩ thuật Divide and Conquer – Giải quyết vấn đề bằng cách chia nhỏ vấn đề ban đầu thành vấn đề nhỏ hơn và sử dụng thuật toán đệ quy để giảivấn đề kích thước lớn tới nhỏ (trên xuống dưới). Tuy nhiên nhược điểm chính của kĩ thuật này là tiêu tốn bộ nhớ sau mỗi lần gọi đệ quy.

- Do vậy, Dynamic Programming hay còn gọi là lập trình động được đề xuất để tránh sử dụng thêm dung lượng. Dynamic Programming cũng gần tương tự kĩ thuật Divide and Conquer với ý tưởng chia nhỏ bài toán thành các vấn đề nhỏ hơn, tuy nhiên DP sử dụng vòng lặp để giải quyết các vấn đề từ nhỏ tới lớn ( dưới lên trên ) khác với DAC.

5.1 Three Basic Examples

- Sau đây chúng ta sẽ tìm hiểu cơ bản về kĩ thuật Dynamic Programming thông qua 3 ví dụ cơ bản và phổ biến.

5.1.1 Coin Row Problem

- Giả sử có một dãy đồng xu có giá trị lần lượt là fi, giá trị mỗi đồng có thể bằng nhau.

* Mục tiêu thiết kế giải thuật để có thể lấy các đồng xu, sao cho tổng giá trị lấy được là lớn nhất, và mỗi đồng xu được lấy không được nằm kế tiếp nhau trên dãy.
* Để giải quyết được vấn đề này, chúng ta cùng đi phân tích hướng giải quyết.
  + Giả sử tổng số đồng xu trên dãy là n
  + Chúng ta sẽ có 2 cách để lấy các đồng xu: lấy các đồng xu có vị trí lẻ hoặc lấy các đồng xu có vị trí chẵn.
    - Cách thứ nhất ( tổng số đồng xu ở vị trí chẵn), Achẵn = fn + A(n-2), với Ak là tổng giá trị các đồng xu lấy được tại vị trí k.
    - Cách thứ hai ( tổng số đồng xu ở vị trí lẻ), Alẻ = A(n-1­).
  + Vậy công thức tính sẽ là Amax = max(Achẵn , Alẻ ) (\*)

* Mã giả:

1. CoinRow(C[1,..n]):
2. // Mục tiêu: áp dụng công thức \* để giải quyết bài toán từ dưới lên.
3. // Input: Một mảng C các đồng xu với các giá trị C[i] là dương.
4. // Ouput: Tổng giá trị nhiều nhất các đồng xu có thể lấy được.
5. F=[] // khởi tạo mảng F có n phần tử.
6. F[0]=0, F[1]=C[1]
7. For i = 2 to n-1 do :
8. F[i] = max(C[i] + F[i-2], F[i-1])
9. Endfor
10. Return F[n-1]

* Code hiện thực thuật toán với python:

def coinRow(C):

    # Mục tiêu: áp dụng công thức \* để giải quyết bài toán từ dưới lên.

    # Input: Một mảng C các đồng xu với các giá trị C[i] là dương.

    # Ouput: Tổng giá trị nhiều nhất các đồng xu có thể lấy được.

    F = []

    F.append(0)

    F.append(C[1])

    for i in range(2, len(C)):

        F.append(max(C[i]+F[i-2], F[i-1]))

    return F[len(C)-1]

* Độ phức tạp thuật toán:
  + T(n) (n)
  + Nếu giải bài toán bằng phép đệ quy kĩ thuật Divide and Conquer thì độ phức tạp thuật toán sẽ vẫn là O(n). Tuy nhiên, nó sẽ tiêu tốn thêm bộ nhớ tại mỗi lần gọi đệ quy.­
* Code test running time:
* Kết quả:

def timmerCoinRow():

    sizeInput = []

    timer = []

    for i in range(2, 101):

        sizeInput.append(i)

        temp = []

        for j in range(i):

            temp.append(random.randint(0, 100))

        start = time.time()

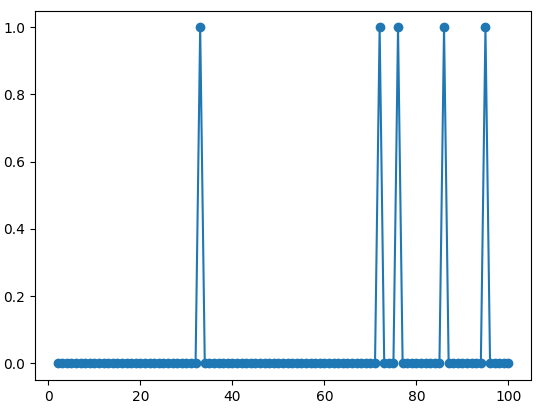
        main.coinRow(temp)

        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()



5.1.2 Change Making Problem:

- Bạn nhận được yêu cầu thiết kế một thuật toán để đổi tiền lẻ.

- Giả sử các đồng xu có giá trị d1, d2, d3, …dm, với d1 = 1. Và số lượng các đồng xu là không giới hạn.

- Thiết kế thuật toán sao cho số lượng tổng các loại đồng xu sau khi đổi là nhỏ nhất.

- Ý tưởng thực hiện :

* Gọi F(n) là tổng số đồng xu it nhất để quy đổi cho n. => F(0) = 0.
* Chúng ta biết rằng, ở mỗi loại đồng xu, để đổi được thì n≥ dj.
* Và sau mỗi lần đổi F(n) cộng thêm 1, và giá trị tiền hiện tại lúc này là n-d­j.
* Do đó, ta có công thức tổng quát: F(n) = min(F(n-dj)) +1.
* Và ta tìm min của F(n-dj) bằng cách thử tất cả trường hợp đổi với từng loại giá trị đồng xu và chọn cách đổi được ít xu nhất.
* Mã giả thuật toán change making:

1. ChangeMaking(D[1,…m],n):
2. // Mục đích: Áp dụng Dynamic Programming để tìm số đồng xu tối thiểu để quy đổi cho n.
3. // Input: Mảng D gồm các giá trị của các loại đồng xu có kích thước m, và giá trị tiền cần đổi n. Và D[1] = 1.
4. // Output: Tổng số đồng xu tối thiểu.
5. F[0]=0
6. For i= 1 to n-1 do:
7. temp= ∞; j=1;
8. While j ≤ m and i ≥ D[j] do:
9. temp = min(F[i-D[j]],temp)
10. j=j+1
11. Endwhile
12. F[i]= temp +1
13. Endfor
14. Return F[n]
15. Endalgorithm

- Code hiện thực thuật toán:

def changeMaking(D, n):

    # Mục đích: Áp dụng Dynamic Programming để tìm số đồng xu tối thiểu để quy đổi cho n

    # Input: Mảng D gồm các giá trị của các loại đồng xu có kích thước m, và giá trị tiền cần đổi n. Và D[1] = 1

    # Output: Tổng số đồng xu tối thiểu

    F = []

    F.append(0)

    for i in range(1, n+1):

        temp = float('inf')

        j = 1

        while j <= len(D)-1 and i >= D[j]:

            temp = min(F[i-D[j]], temp)

            j = j+1

        F.append(temp+1)

    return F[n]

- Độ phức tạp thuật toán:

* + T(n,m) (nm)
* Không gian của thuật toán:
* Code test running time:

def timmerchangeMaking():

    sizeInput = []

    timer = []

    for i in range(3, 103):

        sizeInput.append(i)

        temp = []

        temp.append(1)

        for j in range(2, i):

            temp.append(i)

        trade = random.randint(1, 500)

        start = time.time()

        main.changeMaking(temp, trade)

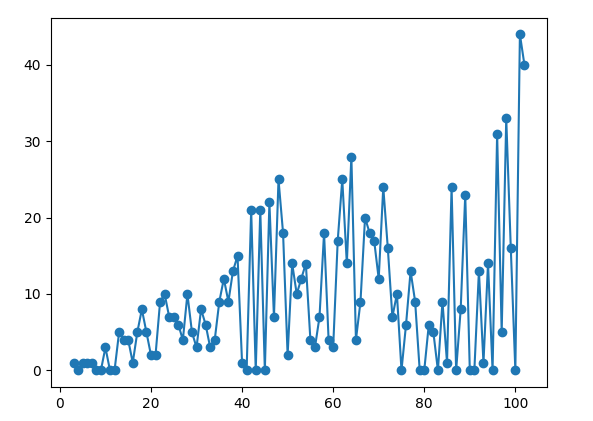
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

* Kết quả:



5.1.3 Coin Collecting Problem:

- Có một bàn cờ có n hàng và m cột. Ở mỗi ô đặt một đồng xu, hãy tính số xu nhiều nhất có thể nhặt được khi đi lần lượt từ ô hàng 1 cột 1 đến ô hàng n cột m. Với điều kiện mỗi lần di chuyển chỉ có thể đi được một ô và từ trên xuống dưới hoặc trái sang phải.

- Mục tiêu của bài toán là tìm max tại F[n][m], với F là tổng số đồng xu nhặt được nhiều nhất tại hàng n cột m.

- Vì chỉ có thể đi được từ trên xuống hoặc trái sang phải nên tại F[n][m] bằng giá trị tại ô cạch trái hoặc ô phía trên và + 1 nếu tại vị trí n, m tồn tại đồng xu hoặc không.

- Do đó ta có công thức sau: F[n][m] = max( F[n-1][m] , F[n][m-1] ) + C[n][m]; với C[n][m] là có hay không đồng xu ở vị trí n, m.

* Mã giả thuật toán như sau:

1. RobotCoinCollection(C[1,..n],[1,…m]):
2. // Mục tiêu: Áp dụng kĩ thuật Dynamic Programming để tìm số lượng đồng xu nhiều nhất nhặt được tại vị trí n m khi đi từ vị trí (1,1). Và chỉ được di chuyển từ trên xuống hoặc trái qua phải.
3. // Input: Một mảng C 2 chiều có kích thước n x m, nếu tại vị trí (i,j) có đồng xu thì C[i][j] = 1, ngược lại = 0.
4. // Output: Số lượng đồng xu nhiều nhất có thể nhặt được.
5. F[n][m] khởi tạo một mảng 2 chiều F có kích thước n x m
6. F[0][0] = C[0][0]
7. For j = 1 to m -1 do :
8. F[0][j] = F[0][j-1] + C[0][j]
9. Endfor
10. For i = 1 to n-1 do:
11. F[i][0]= F[i-1][0]+ C[i][0]
12. For j = 1 to m-1 do:
13. F[i][j] = max( F[i-1][j] , F[i][j-1] ) + C[i][j]
14. Endfor
15. Endfor
16. Return F[n-1][m-1]
17. Endalgorithm

* Code hiện thực thuật toán với python:

def robotCoinCollection(C):

    # Mục tiêu: Áp dụng kĩ thuật Dynamic Programming để tìm số lượng đồng xu nhiều nhất nhặt được

    # tại vị trí n m khi đi từ vị trí (1,1). Và chỉ được di chuyển từ trên xuống hoặc trái qua phải.

    # Input: Một mảng C 2 chiều có kích thước n x m, nếu tại vị trí (i,j) có đồng xu thì C[i][j] = 1, ngược lại = 0.

    # Output: Số lượng đồng xu nhiều nhất có thể nhặt được

    F = [[0 for m in range(len(C[0]))] for n in range(len(C))]

    F[0][0] = C[0][0]

    for j in range(1, len(C[0])):

        F[0][j] = F[0][j-1]+C[0][j]

    for i in range(1, len(C)):

        F[i][0] = F[i-1][0]+C[i][0]

        for j in range(1, len(C[0])):

            F[i][j] = max(F[i-1][j], F[i][j-1])+C[i][j]

    return F[len(C)-1][len(C[0])-1]

- Độ phức tạp thuật toán:

* + T(n,m) = m + n\*m
  + => T(n,m) (nm)
* Không gian của thuật toán:
* Code test running time:

def timmerRobotCoinCollection():

    sizeInput = []

    timer = []

    for i in range(3, 103):

        sizeInput.append(i)

        temp = [[0 for i in range(i)] for j in range(i)]

        for n in range(i):

            for m in range(i):

                temp[n][m] = random.randint(0, 1)

        start = time.time()

        main.robotCoinCollection(temp)

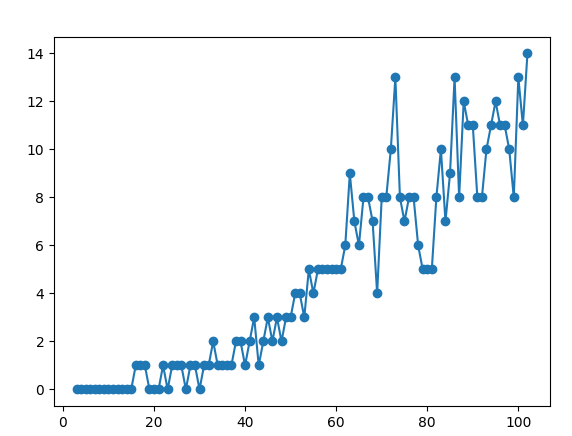
        end = time.time()

        timer.append((end-start)\*1000)  # đơn vị time là 10^3 s

    pylab.plot(sizeInput, timer, 'o-')

    pylab.show()

- Kết quả



5.2 Kết Luận

- Lập trình động là một kỹ thuật để giải quyết các vấn đề chồng chéo các bài toán con.

- Thông thường, các vấn đề con này phát sinh từ sự lặp lại liên quan đến giải pháp cho một vấn đề nhất định với các giải pháp cho các vấn đề con của nó.

- Dynamic Programming giải quyết bài toán bằng các ghi lại kết quả của mỗi vấn đề con vào một bảng và từ đó đưa ra giải pháp cho vấn đề ban đầu.

- Dynamic Programming hoạt động theo một nguyên tắc: giải pháp để giải quyết bài toán phải tối ưu và được tạo thành từ cách giải pháp tối ưu cho từng vấn đề con.

- Dynamic Programming tuy đem lại best thời gian xử lý và không gian tính toán, tuy nhiên nó thực sự khó để thiết kế và không phải ai cũng có thể nghĩ ra được.

CHƯƠNG 6 - BACKTRACKING

- Backtracking – quay lùi là kĩ thuật thiết kế giải thuật dựa trên đệ quy. Backtracking là một kỹ thuật chung cho việc tìm kiếm, xây dựng các giải pháp và loại bỏ các giải pháp không thỏa điều kiện ở mỗi thời điểm bất kì. Hay nói một cách dễ hiểu, thuật toán sẽ thử hết mọi khả năng có thể xảy ra cho đến khi đạt được mục đích của mình.

- Backtracking được áp dụng để giải quyết 3 loại vấn đề chính:

* Decision problem dùng để tìm một giải pháp khả thi cho vấn đề.
* Optimisation Problem dùng để tìm ra giải pháp tốt nhất có thể.
* Enumeration Problem dùng để tìm tất cả các giải pháp khả thi của bài toán.

- Ý tưởng chung:

* Thuật toán sẽ kiểm tra các trường hợp và sẽ có các rằng buộc để kiểm tra, nếu giải pháp không hợp lệ thì sẽ quay ngược trở lại.

- Ý tưởng thiết kế thuật toán:

* Xây dựng một cây các lựa chọn đang thực hiện, được gọi là cây state-space.
* Gốc của cây sẽ là trạng thái trước khi bắt đầu tìm kiếm các giải pháp.
* Các node của cây ở tầng đầu tiên đại diện cho giải pháp vấn đề đầu tiên, và tầng thứ 2 đại diện cho vấn đề thứ 2. ….
* Mỗi node trong cây được coi là Promising (triển vọng) khi nó dẫn đến giải pháp hoàn chỉnh, ngược lại sẽ là NonPromising.
* Nếu node hiện tại không thỏa điều kiện rằng buộc -NonPromising thì thuật toán sẽ quay lùi trở về node cha. Ngược lại, sẽ thêm vào node con tiếp theo cho node có giải pháp hợp lệ.

- Bản chất của Backtracking là quá trình tìm kiếm theo chiều sâu.

- Giờ chúng ta thử xem xét một vài ví dụ để khái quát hóa phương pháp Backtracking.

6.1 Queen Problem

- Bài toán yêu cầu đặt n con hậu (n≥4) trong cờ vua vào một bàn cờ n hàng và n cột, sao n con hậu phải đặt được vào bàn cờ và các con hậu không thể ăn nhau.

- Biết rằng quân cờ hậu có thể ăn trên cùng hàng ngang, dọc và chéo.

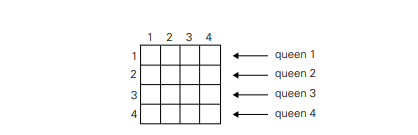
- Ý tưởng:

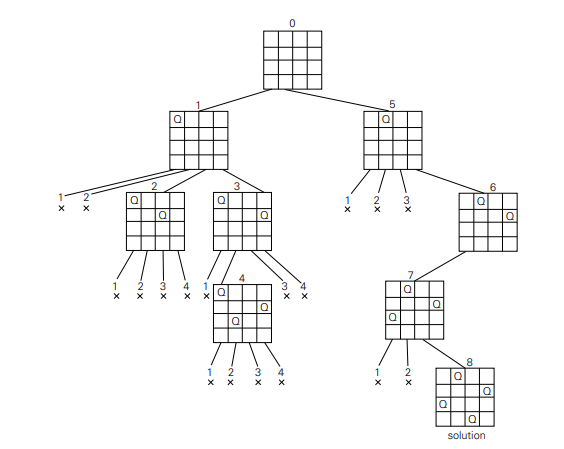
* Thử từng trường hợp ở mỗi quân cờ, nếu không thể đặt con hậu tiếp theo thì sẽ quay lùi lại tại vị trí đặt con hậu trước đó và tiến hành đặt lại. Lặp lại cho tới khi đặt toàn bộ n quân hậu.

- Mã giả:

1. Queen(A,que):
2. // Mục tiêu: sử dụng kĩ thuật backtracking để đặt n queen vào bàn cờ
3. // Input: một mảng A 2 chiều có kích thước nn, và que là queen cần đặt thứ i (bắt đầu từ 0) và tổng queen cần đặt = n.
4. // Output: Tìm vị trí đặt n queen vào mảng 2 chiều ban đầu. Đặt được thì vị trí đó là 1, ngược lại 0, mặc định là 0. Nếu không tìm được cách đặt cho toàn bộ queen thì trả về false.
5. if que ≥ n:
6. return true
7. For i in range(n):
8. If issafe(A,i,que): // tại đây hàm sẽ kiểm tra hàng ngang, dọc chéo xem có an toàn hay ko để đặt cờ.
9. A[i][que]=1
10. If Queen(A,que+1) == true:
11. Return true
12. Endif
13. Else:
14. A[i][que]=0
15. Endif
16. Endfor
17. Return false
18. Endalgorithm

- Ảnh minh họa ( [nguồn](#tailieu) )





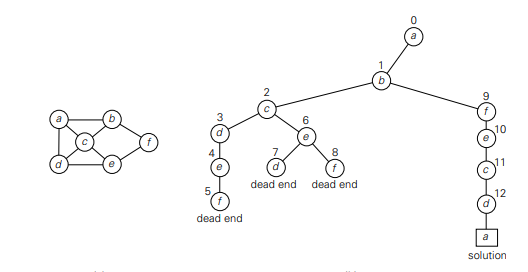
6.2 Hamiltonian Circuit Problem

- Ở ví dụ tiếp theo, chúng ta sẽ tìm đường đi Hamilton từ một đồ thị cho trước.

- Ý tưởng ở bài toán này gần tương tự như bài toán Queen.

- Thuật toán sẽ đi lần lượt bắt đầu từ đỉnh đầu tiên ( mặc định là đỉnh a) và sẽ tìm đường đến đỉnh cuối cùng theo thứ tự bảng chữ cái. Và nếu đó không phải là một giải pháp thì sẽ thực hiện quay lùi lại lần lượt từ đỉnh cuối cùng đến đỉnh đầu tiên và dừng lại khi tìm thấy giải pháp hoàn chỉnh.

- Ảnh minh họa ( [nguồn](#tailieu) )



6.3 Subset Sum Problem

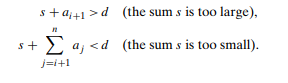
- Ví dụ cuối cùng của chúng ta đó là bài toán tìm các tập con của một tập cho trước, sao cho tổng các phần tử ở mỗi tập con phải bằng giá trị k cho trước.

- Giả sử các phần tử của tập ban đầu được sắp xếp tăng dần để cho thuận tiện.

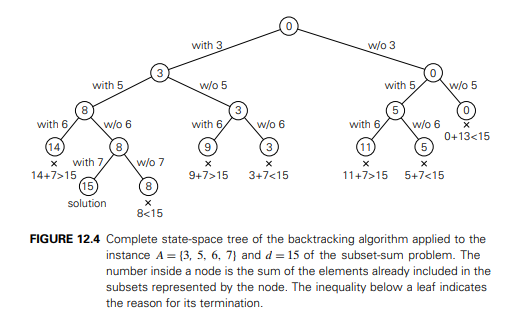
- Ý tưởng: Bài toán sẽ vẫn áp dụng kỹ thuật backtracking với việc xây dựng một cây nhị phân, với mỗi node là tổng hiện tại của tập con và node con bên trái sẽ là thêm vào phần tử tiếp theo và node con bên phải là bỏ qua phần tử con tiếp theo.

- Sau chạy hết các phần tử trong tập hợp ban đầu thì sẽ kiểm tra node cuối cùng của mỗi nhánh, giá trị nào bằng với k thì sẽ suy ngược lên để tìm ra từng tập con thỏa điều kiện ban đầu.

- Nếu node thỏa điều kiện sau thì sẽ bỏ qua không tính tiếp và quay lùi.



- Ảnh minh họa ( [nguồn](#tailieu) )



6.4 General Remarks

- Hầu hết các thuật toán Backtracking phù hợp với các bài toán có đầu ra là một n-tuple.

- Các thuật toán Backtracking thường được áp dụng để giải quyết các bài toán tổ hợp khó mà không có thuật toán chuyên biệt hiệu quả.

- Backtracking cực kì chậm bởi việc phải thử với từng trường hợp, tuy nhiên ở bài toán có kích thước không đáng kể, thời gian thực thi có thể chấp nhận được.

- Backtracking sử dụng để quy và việc xây dựng cây state-space của nó làm tiêu tốn rất nhiều bộ nhớ.

- Mã giả:

1. Backtracking(X):
2. // Công thức chung của kỹ thuật Backtracking
3. // Input: Con trỏ X chứa các giải pháp có kích thước k.
4. // Output: Tất cả các giải pháp cho vấn đề n-tuple S.
5. If X[1,…k] là giải pháp hợp lý :
6. In ra X[1,…k]
7. Else:
8. For each element x Si+1  phù hợp với X[1,…k] và điều kiện :
9. X[i+1]=x
10. Backtracking(X[1,…i+1])
11. Endforeach
12. Endif
13. Endalgorithm

**TÀI LIỆU THAM KHẢO**

**Tiếng Anh**

Introduction to The Design and Analysis of Algorithms – Third Edition. Anany Levitin ( Villanova University ).

BÁO CÁO MỖI TUẦN

|  |  |
| --- | --- |
| **Tuần** | **Nội Dung Làm** |
| **1**  **(12-18/10)** | * **Hoàn thành Phần Introduction và 2 kỹ thuật Brute Force, Divide and Conquer** * **Thực hiện Code demo và Viết hàm Running time** |
| **2**  **(19-25/10)** | * **Hoàn thành 2 kỹ thuật còn lại Greedy và Dynamic** * **Thực hiện Code demo và Viết hàm Running time** * **Hoàn thành phần tự học Backtracking** |