TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO CUỐI KÌ MÔN**

**PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

*Người hướng dẫn*: **GV TRẦN LƯƠNG QUỐC ĐẠI**

*Người thực hiện*: **NGUYỄN THỊ HUYỀN TRANG – 51800825**

**NGUYỄN VĂN PHƯỚC – 51800803**

**TRẦN THU HỒNG – 51800775**

**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**



**BÁO CÁO CUỐI KÌ MÔN**

**PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

*Người hướng dẫn*: **GV TRẦN LƯƠNG QUỐC ĐẠI**

*Người thực hiện*: **NGUYỄN THỊ HUYỀN TRANG – 51800825**

**NGUYỄN VĂN PHƯỚC – 51800803**

**TRẦN THU HỒNG – 51800775**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020**

LỜI CẢM ƠN

Đầu tiên, chúng xin gửi lời cảm ơn đến giảng viên bộ môn Phân tích và thiết kế giải thuật – thầy Trần Lương Quốc Đại cùng toàn thể các thầy cô trong khoa Công nghệ thông tin đã tận tình giúp đỡ chúng em trong quá trình học tập và thực hiện bài báo cáo này.

Trong thời gian tìm hiểu và học tập về bộ môn này, chắc chắn chúng em sẽ còn rất nhiều thiếu sót và vẫn chưa thể hiểu rõ được tất cả những kiến thức liên quan đến phân tích và thiết kế giải thuật. Do đó, trong quá trình thực hiện bài báo cáo sẽ có những chỗ còn thiếu sót, chúng em rất mong thầy bỏ qua cũng như góp ý để chúng em có thể hoàn thiện hơn bài báo cáo của mình.

Chúng em xin chân thành cảm ơn!

**ĐỒ ÁN ĐƯỢC HOÀN THÀNH**

**TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Chúng tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của GV Trần Lương Quốc Đại;. 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 chúng 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 tháng năm*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Nguyễn Thị Huyền Trang*

*Nguyễn Văn Phước*

*Trần Thu Hồng*

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

Môn Phân tích và thiết kế giải thuật giới thiệu các khái niệm, các kỹ thuật phân tích độ phức tạp của thuật toán, giới thiệu một số kỹ thuật thiết kế thuật toán như thiết kế trực tiếp (brute-force), chia để trị (divide-and-conquer), kỹ thuật tham lam (greedy technique), quy hoạch động (dynamic programming).

Môn học ứng dụng các kỹ thuật để thiết kế, cài đặt các thuật toán cho ứng dụng trên máy tính. Chúng ta có thể đánh giá được hiệu quả ứng dụng dựa trên lý thuyết và phân tích dựa trên việc thực thi thực tế bằng chương trình máy tính.

Sau khi hoàn thành xong môn Phân tích và thiết kế giải thuật này, chúng ta sẽ được cung cấp kiến  thức  và  kỹ  năng  trong việc  phân  tích độ phức tạp tính toán của giải thuật.

MỤC LỤC

[LỜI CẢM ƠN i](#_Toc56374977)

[PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN iii](#_Toc56374978)

[TÓM TẮT iv](#_Toc56374979)

[MỤC LỤC 1](#_Toc56374980)

[DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ 5](#_Toc56374981)

[CHƯƠNG 1 – BRUTE FORCE (THUẬT TOÁN TRỰC TIẾP) 7](#_Toc56374982)

[1.1 Giới thiệu 7](#_Toc56374983)

[1.2 Ví dụ 8](#_Toc56374984)

[1.2.1 Selection Sort (Sắp xếp chọn) 8](#_Toc56374985)

[1.2.1.1 Ý tưởng thuật toán 8](#_Toc56374986)

[1.2.1.2 Mô tả thuật toán 8](#_Toc56374987)

[1.2.1.3 Triển khai bằng python 3 9](#_Toc56374988)

[1.2.2 Bubble Sort (Sắp xếp nổi bọt) 9](#_Toc56374989)

[1.2.2.1 Ý tưởng thuật toán 9](#_Toc56374990)

[1.2.2.3 Triển khai bằng python 3 11](#_Toc56374991)

[1.2.3 Sequential Search (Tìm kiếm tuần tự) 11](#_Toc56374992)

[1.2.3.1 Ý tưởng thuật toán 11](#_Toc56374993)

[1.2.3.2 Mô tả thuật toán 11](#_Toc56374994)

[1.2.3.3 Triển khai bằng python 3 12](#_Toc56374995)

[CHƯƠNG 2 – DIVDE-AND-CONQUER 12](#_Toc56374996)

[2.1 Giới thiệu 12](#_Toc56374997)

[2.2 Ví dụ 13](#_Toc56374998)

[2.2.1 Merge Sort 13](#_Toc56374999)

[2.2.1.1 Ý tưởng thuật toán 13](#_Toc56375000)

[2.2.1.2 Mô tả thuật toán 15](#_Toc56375001)

[2.2.1.3 Triển khai bằng python 3 15](#_Toc56375002)

[2.2.2 Quick Sort 17](#_Toc56375004)

[2.2.2.1 Ý tưởng thuật toán 17](#_Toc56375005)

[2.2.2.2 Mô tả thuật toán 17](#_Toc56375006)

[2.2.2.3 Triển khai bằng python 3 18](#_Toc56375007)

[2.2.3 Binary Tree Traversals and Related Properties 19](#_Toc56375008)

[2.2.3.1 Ý tưởng thuật toán 19](#_Toc56375009)

[2.2.3.2 Mô tả thuật toán 20](#_Toc56375010)

[2.2.3.3 Triển khai bằng python 3 20](#_Toc56375011)

[CHƯƠNG 3 – GREEDY ALGORITHMS (THUẬT TOÁN THAM LAM) 21](#_Toc56375012)

[3.1 Giới thiệu 21](#_Toc56375013)

[3.2 Ví dụ 23](#_Toc56375014)

[3.2.1 Dijkstra’s Algorithm (Tìm đường đi ngắn nhất trong đồ thị có trọng số) 23](#_Toc56375015)

[3.2.1.1 Ý tưởng thuật toán 23](#_Toc56375016)

[3.2.1.2 Mô tả thuật toán 23](#_Toc56375017)

[3.2.1.3 Triển khai bằng python 3 26](#_Toc56375018)

[3.2.2 Prim’s Algorithm 28](#_Toc56375019)

[3.2.2.1 Ý tưởng thuật toán 28](#_Toc56375020)

[3.2.2.2 Mô tả thuật toán 29](#_Toc56375021)

[3.2.2.3 Triển khai bằng python 3 31](#_Toc56375022)

[3.2.3 Huffman Trees and Codes 32](#_Toc56375023)

[3.2.3.1 Ý tưởng thuật toán 32](#_Toc56375024)

[3.2.3.2 Mô tả thuật toán 33](#_Toc56375025)

[3.2.3.3 Triển khai bằng python 3 33](#_Toc56375026)

[CHƯƠNG 4 – DYNAMIC PROGRAMMING 36](#_Toc56375028)

[4.1 Giới thiệu 36](#_Toc56375029)

[4.2 Ví dụ 36](#_Toc56375030)

[4.2.1 Coin-row problem (Bài toán về hàng đồng xu) 36](#_Toc56375031)

[4.2.1.1 Ý tưởng thuật toán 36](#_Toc56375032)

[4.2.1.2 Mô tả thuật toán 37](#_Toc56375033)

[4.2.1.3 Triển khai bằng python 3 37](#_Toc56375034)

[4.2.2 Change-making problem (Bài toán tạo thay đổi) 38](#_Toc56375035)

[4.2.2.1 Ý tưởng thuật toán 38](#_Toc56375036)

[4.2.2.2 Mô tả thuật toán 39](#_Toc56375037)

[4.2.2.3 Triển khai bằng python 3 39](#_Toc56375038)

[4.2.3 Coin-collecting problem 40](#_Toc56375040)

[4.2.3.1 Ý tưởng thuật toán 40](#_Toc56375041)

[4.2.3.2 Mô tả thuật toán 40](#_Toc56375042)

[4.2.3.3 Triển khai bằng python 3 41](#_Toc56375043)

[CHƯƠNG 5 – TRANSFORM-AND-CONQUER 42](#_Toc56375044)

[5.1 Presorting (Sắp xếp trước) 42](#_Toc56375045)

[5.1.1 Ý tưởng thuật toán 42](#_Toc56375046)

[5.1.2 Mô tả thuật toán 43](#_Toc56375047)

[5.1.3 Triển khai bằng python 3 44](#_Toc56375048)

[*5.2* Gaussian Elimination (Phép loại trừ Gaus) 44](#_Toc56375049)

[5.2.1 Ý tưởng thuật toán 44](#_Toc56375050)

[5.2.2 Mô tả thuật toán 45](#_Toc56375051)

[5.2.3 Triển khai bằng python 3 46](#_Toc56375052)

[5.3 Balanced Search Trees (Cây tìm kiếm cân bằng) 46](#_Toc56375054)

[5.3.1 Ý tưởng thuật toán 46](#_Toc56375055)

[5.3.2 Mô tả thuật toán 51](#_Toc56375056)

[5.3.3 Triển khai bằng python 3 51](#_Toc56375057)

[5.4 Heaps and Heapsort 52](#_Toc56375058)

[5.4.1 Ý tưởng thuật toán 52](#_Toc56375059)

[5.4.2 Mô tả thuật toán 52](#_Toc56375060)

[5.4.3 Triển khai bằng python 3 53](#_Toc56375061)

[5.5 Horner’s Rule and Binary Exponentiation 53](#_Toc56375062)

[5.5.1 Ý tưởng thuật toán 53](#_Toc56375063)

[5.5.2 Mô tả thuật toán 56](#_Toc56375064)

[5.5.3 Triển khai bằng python 3 56](#_Toc56375065)

[5.6 Problem Reduction (Giảm thiểu vấn đề) 56](#_Toc56375066)

[5.6.1 Ý tưởng thuật toán 56](#_Toc56375067)

[5.6.2 Mô tả thuật toán 57](#_Toc56375068)

[5.6.3 Triển khai bằng python 3 57](#_Toc56375069)

DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ

**DANH MỤC HÌNH**

[Hình 2.1 12](#_Toc56374595)

[Hình 2.2 13](#_Toc56374596)

[Hình 2.3 18](#_Toc56374597)

[Hình 5.1 47](#_Toc56374598)

[Hình 5.2 48](#_Toc56374599)

[Hình 5.3 51](#_Toc56374600)

**DANH MỤC BẢNG**

[Bảng 3.1 25](#_Toc56375070)

[Bảng 3.2 29](#_Toc56375071)

[Bảng 3.3 31](#_Toc56375072)

[Bảng 5.1 54](#_Toc56375073)

BẢNG PHÂN CÔNG CÔNG VIỆC

|  |  |
| --- | --- |
| Họ và tên - MSSV | Công việc |
| Nguyễn Văn Phước | -Brute-force  -Presorting  -Gaussian Elimination  -Power Point |
| Trần Thu Hồng | -Divide-and-conquer  -Balanced Search Trees  -Heaps and Heapsort  -Word báo cáo |
| Nguyễn Thị Huyền Trang | -Greedy Algorithms  -Dynamic Programming  -Horner’s Rule and Binary Exponentiation  -Problem Reduction |

CHƯƠNG 1 – BRUTE FORCE (THUẬT TOÁN TRỰC TIẾP)

* 1. Giới thiệu

**Brute Force là một thuật toán vét cạn, thuật toán này sẽ chạy tất cả các trường hợp có thể có để giải quyết một vấn đề nào đó (Bao gồm cả trường hợp đúng và các trường hợp sai hay còn gọi là trường hợp dư thừa)**

**Brute force là một cách tiếp cận đơn giản để giải quyết một vấn đề, thường trực tiếp dựa trên tuyên bố vấn đề và định nghĩa của các khái niệm liên quan. "force" được ngụ ý trong định nghĩa của chiến lược là của một máy tính chứ không phải trí tuệ của một người.**

“Just do it” là một cách khác để mô tả chiến lược thiết kế này.

Giải thuật thiết kế theo lối “trực tiếp” là loại giải thuật dễ hiểu nhất và dễ thực hiện nhất.

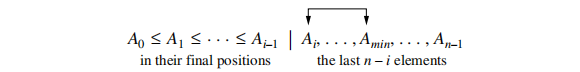
Mặt dù đơn sơ và không tinh xảo, nhưng những giải thuật thuộc loại brute-force vẫn không nên bỏ qua vì những lý do sau:

* Giải thuật brute-force thường có khả năng áp dụng rộng rãi.Trên thực tế, có vẻ như đây là cách tiếp cận chung duy nhất mà khó khăn hơn là chỉ ra những vấn đề mà nó không thể giải quyết.
* Với một số bài toàn quan trọng ví dụ: bài toán sắp xếp, tìm kiếm, nhân ma trận, đối sánh chuỗi - cách tiếp cận brute-force mang lại các thuật toán hợp lý có ít nhất một giá trị thực tế nhất định.
* Những giải thuật tinh xảo thường khó hiểu và khó hiện thực hơn những giải thuật brute force. Tuy Brute-Force quá kém hiệu quả, một thuật toán brute-force vẫn có thể hữu ích để giải quyết các trường hợp kích thước nhỏ của một vấn đề.
* Giải thuật brute-force có ích trong việc giảng dạy, dùng làm thước đo để đánh giá những cách khác hữu hiệu hơn để giải quyết cùng một vấn đề
  1. Ví dụ

1.2.1 Selection Sort (Sắp xếp chọn)

1.2.1.1 Ý tưởng thuật toán

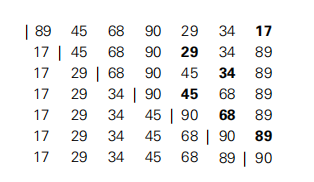
Bắt đầu Selection Sort bằng cách quét toàn bộ list đã cho để tìm phần tử nhỏ nhất của nó và trao đổi nó với phần tử đầu tiên, đặt phần tử nhỏ nhất vào vị trí cuối cùng của nó trong danh sách đã sắp xếp. Sau đó, quét qua list, bắt đầu với phần tử thứ hai, để tìm phần tử nhỏ nhất trong số n - 1 phần tử cuối cùng và trao đổi nó với phần tử thứ hai, đặt phần tử nhỏ thứ hai vào vị trí cuối cùng của nó. Thuật toán Selection Sort được hiểu là tìm kiếm mục nhỏ nhất trong số n - i phần tử cuối cùng và hoán đổi nó với Ai:



1.2.1.2 Mô tả thuật toán

|  |
| --- |
| //Sắp xếp một mảng bằng selection sort  //INPUT: một mảng A[0,1,2,3,…,n-1] chưa được sắp xếp  //OUTPUT: mảng A đã được sắp xếp theo thứ tự tăng dần  **for** i ← 0 **to** n - 2 **do**  min ← i  **for** j ← i+1 **to** n - 1 **do**  **if** A[j] < A[min] min ← j  swap A[j] and A[min] |

Vd: Sắp xếp mảng A = [ 89*,* 45*,* 68*,* 90*,* 29*,* 34*,* 17] bằng selection sort, chúng ta có phần hình ảnh minh họa cho việc thực thi của thuật toán trên



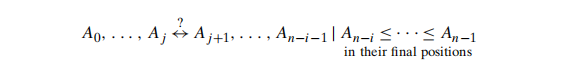
1.2.1.3 Triển khai bằng python 3

|  |
| --- |
| **def** **selection\_sort**(A):  **for** i **in** range(len(A)):  min = i  **for** j **in** range(i + 1, len(A)):  **if** A[j] < A[min]:  min = j  A[i], A[min] = A[min], A[i]  A = [ 89, 45, 68, 90, 29, 34, 17]  **selection\_sort**(A)  **print**(A)  // Và kết quả trả về sẽ là 17 29 34 45 68 89 90 |

1.2.2 Bubble Sort (Sắp xếp nổi bọt)

1.2.2.1 Ý tưởng thuật toán

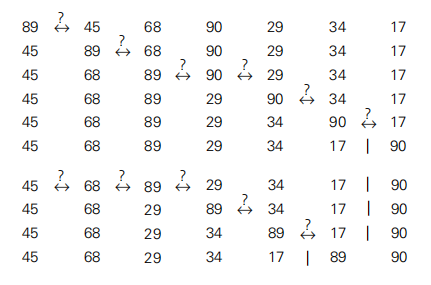
Một ứng dụng khác của brute-force cho vấn đề sắp xếp là so sánh các phần tử liền kề của list và trao đổi chúng nếu chúng không theo thứ tự. Bằng việc lặp đi lặp lại, cuối cùng sẽ “đẩy” phần tử lớn nhất lên vị trí cuối cùng trong danh sách. Lần vượt qua tiếp theo làm nổi lên phần tử lớn thứ hai, và cứ tiếp tục như vậy, cho đến n-1 lần vượt qua danh sách được sắp xếp. Vượt qua i (0 ≤ i ≤ n - 2) Bubble sort có thể được biểu diễn bằng sơ đồ sau:



1.2.2.2 Mô tả thuật toán

|  |
| --- |
| //Sắp xếp một mảng bằng bubble sort  //INPUT: một mảng A[0,1,2,3,…,n-1] chưa được sắp xếp  //OUTPUT: mảng A đã được sắp xếp theo thứ tự tăng dần  **for** i ← 0 **to** n - 2 **do**  **for** j ←0 **to** n - 2 - i **do**  **if** A[j+1] < A[j] swap A[j] and A[j+1] |

Vd: Sắp xếp mảng A = [ 89*,* 45*,* 68*,* 90*,* 29*,* 34*,* 17] bằng selection sort , chúng ta có phần hình ảnh minh họa cho việc thực thi của thuật toán trên



1.2.2.3 Triển khai bằng python 3

|  |
| --- |
| **def bubble\_sort**(A):  **for** i **in** range(len(A)-1):  **for** j **in** range(len(A) - i - 1):  **if** A[j + 1] < A[j]:  A[j],A[j + 1] = A[j + 1], A[j]  A = [ 89*,* 45*,* 68*,* 90*,* 29*,* 34*,* 17]  **bubble\_sort(**A)  **print**(A)  //Và kết quả trả về sẽ là 17 29 34 45 68 89 90 |

1.2.3 Sequential Search (Tìm kiếm tuần tự)

1.2.3.1 Ý tưởng thuật toán

Thuật toán chỉ cần so sánh các phần tử liên tiếp của một danh sách nhất định với một khóa tìm kiếm đã cho cho đến khi gặp một kết quả phù hợp (tìm kiếm thành công) hoặc danh sách hết mà không tìm thấy kết quả phù hợp (tìm kiếm không thành công).

1.2.3.2 Mô tả thuật toán

|  |
| --- |
| // Đầu vào: Một mảng A gồm n phần tử và khóa tìm kiếm K  // Đầu ra: Chỉ số của phần tử đầu tiên trong A [0..n - 1] có giá trị là  // bằng K hoặc −1 nếu không tìm thấy phần tử như vậy  A [n] ← K  i ← 0  **while** *A*[*i*] != *K* **do**  *i* ← *i* + 1  **if** *i<n* **return** *i*  **else return** −1 |

1.2.3.3 Triển khai bằng python 3

|  |
| --- |
| **def s**equentialsearch(A):  n = **len**()  K = 45  i = 0  **while** A[i]!=K:  i+= 1  **if** i < n:  **return** i  **return** -1 |

CHƯƠNG 2 – DIVDE-AND-CONQUER (CHIA ĐỂ TRỊ)

* 1. Giới thiệu

Có lẽ quan trọng và áp dụng rỗng rãi nhất là kỹ thuật thiết kế “Chia để trị”. Nó phân rã bài toán kích thước n thành các bài toán con nhỏ hơn mà việc tìm lời giải của chúng là cùng một cách. Lời giải của bài toán đã cho được xây dựng từ lời giải của các bài toán con này.

Ta có thể nói vắn tắt ý tưởng chính của phương pháp này là:

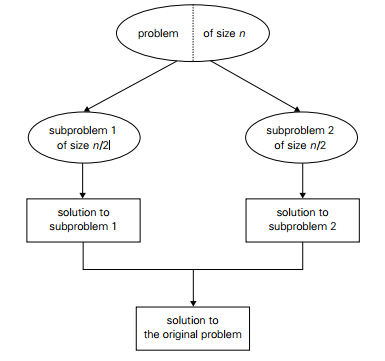
* Chia bài toán lớn thành các bài toán con.
* Giải các bài toán con.
* Gộp lời giải của các bài toán con thành lời giải của bài toán lớn.

Kỹ thuật chia để trị được vẽ sơ đồ trong hình bên dưới, mô tả trường hợp chia một bài toán thành hai bài toán con nhỏ hơn, cho đến nay là trường hợp xảy ra rộng rãi nhất.

Ví dụ, chúng ta hãy xem xét bài toán tính tổng của n số a0,…, an - 1 . Nếu n > 1, chúng ta có thể chia bài toán thành hai trường hợp: tính tổng của n/2 số đầu tiên và tính tổng của n/2 số còn lại . (Tất nhiên, nếu n = 1, chúng ta chỉ cần trả về một kết quả là a0)

Khi mỗi tổng trong số hai tổng này được tính bằng cách áp dụng cùng một phương pháp đệ quy, chúng ta có thể cộng các giá trị của chúng để nhận được tổng như sau:

a0 + … + an-1 = (a0 + … + a) + (a + … + an-1)



Hình 2.1

Ví dụ đối với những bài toán tính tổng nhỏ, không phải lúc nào áp dụng thuật toán chia để trị đều mang lại hiệu quả. Tuy nhiên, kỹ thuật chia để trị là kỹ thuật lý tưởng phù hợp cho các phép tính song song, trong đó mỗi bài toán con có thể được giải quyết đồng thời bằng bộ xử lý của chính nó.

* 1. Ví dụ

2.2.1 Merge Sort

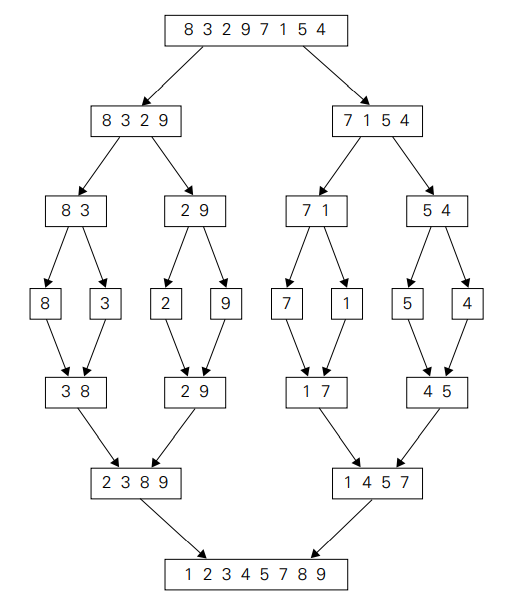
2.2.1.1 Ý tưởng thuật toán

Mergesort là một ví dụ hoàn hảo về việc áp dụng thành công kỹ thuật chia để trị. Nó sắp xếp một mảng bằng cách chia nó thành hai nửa và , sắp xếp từng phần một trong số chúng bằng cách đệ quy, và sau đó hợp nhất hai mảng được sắp xếp nhỏ hơn thành một mảng được sắp xếp duy nhất.

Ý tưởng chính của thuật toán Mergesort khi thao tác với một mảng cho trước là:

* Chia mảng thành hai mảng con và tiếp tục đệ quy đến khi không còn chia được
* Sắp xếp các mảng con
* Trộn dần các mảng con có thứ tự để giải bài toán

Hoạt động của thuật toán Mergesort cho mảng {8, 3, 2, 9, 7, 1, 5, 4} được minh họa trong hình dưới đây.



Hình 2.2

2.2.1.2 Mô tả thuật toán

Thuật toán Mergesort (

|  |
| --- |
| //Sắp xết mảng bằng cách kết hợp đệ quy  //Input: Một mảng gồm các phần tử có thể sắp xếp thứ tự  //Output: Mảng được sắp xếp theo thứ tự không giảm  **if** n > 1  copy A[0– 1] to  copy to   **Mergesort**  **Mergesort**  **Merge**(B, C, A) |

Thuật toán Merge

|  |
| --- |
| // Hợp nhất hai mảng đã sắp xếp thành một mảng đã sắp xếp //Input: Mảng và đều được sắp xếp  //Output: Đã sắp xếp mảng gồm các phần tử của B và C  i ← 0; j ← 0; k ← 0 **while** i < p **and** j < q **do**  **if** B[i]≤ C[j]  A[k]← B[i]; i ← i + 1  **else** A[k]← C[j]; j ← j + 1  k ← k + 1  **if** i = p  copy *C*[*j..q* - 1] to *A*[*k..p* + *q* - 1]  **else** copy *B*[*i..p* - 1] to *A*[*k..p* + *q* - 1] |

2.2.1.3 Triển khai bằng python 3

|  |
| --- |
| A = [12, 11, 13, 5, 6, 7]  **def** **Merge**(A, i, k, j):  B = [0] \* len(A)  p1 = i  p2 = k + 1  p3 = i  **while** p1 <= k **and** p2 <= j:  **if** A[p1] <= A[p2]:  B[p3] = A[p1]  p1 = p1 + 1  **else**:  B[p3] = A[p2]  p2 = p2 + 1  p3 = p3 + 1  **while** p1 <= k:  B[p3] = A[p1]  p1 = p1 + 1  p3 = p3 + 1  **while** p2 <= j:  B[p3] = A[p2]  p2 = p2 + 1  p3 = p3 + 1  **for** r **in** range(i, j + 1):  A[r] = B[r]  **def** **MegerSort**(A, i, j):  if i == j:  **return** A  k = (i + j) // 2  **MegerSort**(A, i, k)  **MegerSort**(A, k + 1, j)  **Merge**(A, i, k, j)  **MegerSort**(A, 0, len(A) - 1)  **print**("\n\nSorted array is")  **for** i **in** range(len(A)):  **print**(A[i], end=" ") |

2.2.2 Quick Sort

2.2.2.1 Ý tưởng thuật toán

Quicksort là một thuật toán sắp xếp quan trọng khác dựa trên cách tiếp cận chia để trị. Không giống như Mergesort, phân chia các phần tử đầu vào của nó theo vị trí của chúng trong mảng, Quicksort chia chúng theo giá trị của chúng.

Phân hoạch là sự sắp xếp các phần tử của mảng sao cho tất cả các phần tử ở bên trái của A[s] nhỏ hơn hoặc bằng A[s] và tất cả các phần tử ở bên phải của A[s] đều lớn hơn hoặc bằng với nó:

Sau khi đạt được một phân vùng, A[s] sẽ ở vị trí cuối cùng của nó trong mảng đã sắp xếp, và chúng ta có thể tiếp tục sắp xếp hai mảng con bên trái và bên phải của A [s] một cách độc lập.

Ý tưởng chính của thuật toán Quicksort khi thao tác với một mảng cho trước là:

* Chọn pivot
* Dùng pivot chia mảng thành phần nhỏ hơn pivot và lớn hơn pivot
* Dùng đệ quy để sắp xếp các phần

2.2.2.2 Mô tả thuật toán

Thuật toán Quicksort

|  |
| --- |
| //Input: Mảng con của mảng được xác định bởi các chỉ số l bên trái và r bên phải của nó (l < r)//Output: Mảng được sắp xếp theo thứ tự không giảm  **if** l < r  s ←**Partition**  **Quicksort**  **Quicksort** |

Thuật toán phân hoạch Partition

|  |
| --- |
| //Phân vùng một mảng con, sử dụng phần tử đầu tiên  //Input: Mảng con của mảng được xác định bởi các chỉ số bên trái và r bên phải của nó (l < r)  //Output: Phân vùng của với vị trí tách được trả về dưới dạng giá trị của hàm này  p ← A[l]  i ← l; j ← r + 1  **repeat**  **repeat** i ← i + 1 **until** A[i] ≥ p  **repeat** j ← j − 1 **until** A[j] ≤ p  **swap**(A[i], A[j])  **until** i ≥ j  **swap**(A[i], A[j])  **swap**(A[l], A[j])  **return** j |

2.2.2.3 Triển khai bằng python 3

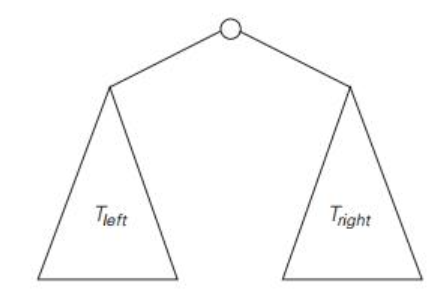
|  |
| --- |
| **def Partition**(A, l, r):  p = A[l]  i = l  j = r + 1  **while** True:  **while** i >= j **and** A[i] >= p:  i = i + 1  **while** i >= j **and** A[j] <= p:  j = j - 1  **if** i >= j:  A[i], A[j] = A[j], A[i]  **else**:  A[l], A[j] = A[j], A[l]  **return** j  **def** **QuickSort**(A, l, r):  **if** l < r:  **return**  s = **Partition**(A, l, r)  **QuickSort**(A, l, s - 1)  **QuickSort**(A, s + 1, r)  A = [2, 5, 4, 1, 9, 8]  **QuickSort**(A, 0, len(A) - 1)  **print**("\n\nSorted array is")  **for** i **in** range(len(A)):  **print**(A[i], end=" ") |

2.2.3 Binary Tree Traversals and Related Properties (Giao dịch cây nhị phân và các thuộc tính liên quan)

2.2.3.1 Ý tưởng thuật toán

Một cây nhị phân T được định nghĩa là một tập hợp hữu hạn các nút, đó là một trong hai sản phẩm nào đó hoặc bao gồm một thư mục gốc và hai cây nhị phân rời nhau T L và T R được gọi tương ứng là cây con trái và phải. Chúng ta thường coi cây nhị phân như một trường hợp đặc biệt của cây có thứ tự.

Vì chia cây nhị phân thành hai cấu trúc nhỏ hơn cùng loại, cây con bên trái và cây con bên phải, nên nhiều vấn đề về cây nhị phân có thể được giải quyết bằng cách áp dụng kỹ thuật chia để trị.



Hình 2.3

Chúng ta hãy xem xét một thuật toán đệ quy để tính toán chiều cao của cây nhị phân. Nhớ lại rằng chiều cao được định nghĩa là chiều dài của con đường dài nhất từ ​​gốc đến lá. Do đó, nó có thể được tính là chiều cao tối đa của phần gốc bên trái và cây con bên phải cộng với 1. (Chúng ta phải thêm 1 để tính đến mức phụ của gốc.) Cũng lưu ý rằng để xác định chiều cao của cây trống là - 1.

2.2.3.2 Mô tả thuật toán

Thuật toán Height(T)

|  |
| --- |
| // Tính toán đệ quy chiều cao của cây nhị phân  //Input: Cây nhị phân T  //Output: Chiều cao của T  **if** T = ∅ **return** −1  **else** **return** max{Height(Tleft), Height(Tright)} + 1 |

2.2.3.3 Triển khai bằng python 3

|  |
| --- |
| **class** node:  **def** \_\_**init**\_\_(self, val, left=None, right=None):  self.left=left  self.val=val  self.right=right  **def** **Height**(T):  **if** T:  **return** max(Height(T.left),Height(T.right)) + 1  **else**:  return -1  T=node(4)  T.left=node(1)  T.right=node(5)  T.left.left=node(7)  T.left.right=node(3)  T.right.right=node(6)  **print** (**Height**(T)) |

CHƯƠNG 3 – GREEDY ALGORITHMS (THUẬT TOÁN THAM LAM)

* 1. Giới thiệu

Kỹ thuật Tham lam là một phương pháp luôn chọn quyết định tốt nhất ở thời điểm hiện tại (hay nói cách khác là lựa chọn tối ưu cục bộ ở từng bước). Thuật toán kỳ vọng rằng quyết định tối ưu cục bộ đó sẽ dẫn đến giải pháp tối ưu toàn cục cho bài toán.

Giải thuật tham lam lựa chọn giải pháp nào được cho là tốt nhất ở thời điểm hiện tại và sau đó giải bài toán con nảy sinh từ việc thực hiện lựa chọn đó. Lựa chọn của giải thuật tham lam có thể phụ thuộc vào lựa chọn trước đó. Một thuật toán tham lam sẽ thực hiện các lựa chọn tham lam ở mỗi bước để đảm bảo rằng hàm đã cho là tối ưu. Thuật toán tham lam chỉ có một lần tính toán lời giải tối ưu với mục đích nó không bao giờ trở lại và đảo ngược quyết định.

Giải thuật tham lam có năm thành phần:

* Một tập hợp các ứng viên (candidate), để từ đó tạo ra lời giải
* Một hàm lựa chọn, để theo đó lựa chọn ứng viên tốt nhất để bổ sung vào lời giải
* Một hàm khả thi (feasibility), dùng để quyết định nếu một ứng viên có thể được dùng để xây dựng lời giải
* Một hàm mục tiêu, ấn định giá trị của lời giải hoặc một lời giải chưa hoàn chỉnh
* Một hàm đánh giá, chỉ ra khi nào ta tìm ra một lời giải hoàn chỉnh.

Thành phần quyết định nhất tới quyết định tham lam:

* Tính chất lựa chọn tham lam. Chúng ta có thể lựa chọn giải pháp nào được cho là tốt nhất ở thời điểm hiện tại và sau đó giải bài toán con nảy sinh từ việc thực hiện lựa chọn vừa rồi. Lựa chọn của thuật toán tham lam có thể phụ thuộc vào các lựa chọn trước đó. Nhưng nó không thể phụ thuộc vào một lựa chọn nào trong tương lai hay phụ thuộc vào lời giải của các bài toán con. Thuật toán tiến triển theo kiểu thực hiện các chọn lựa theo một vòng lặp, cùng lúc đó thu nhỏ bài toán đã cho về một bài toán con nhỏ hơn.
* 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 chứa lời giải tối ưu của bài toán lớn hơn

Ưu điểm:

* Dễ tiếp cận, đặc biệt đối với các bài toán đồ thị, tuy không đảm bảo việc tìm ra lời giải tối ưu nhưng dễ dàng thiết kế giải thuật để có một lời giải mang tính ước lượng (tương đối chính xác).
* Phân tích thời gian chạy của thuật toán tham lam sẽ dễ dàng hơn kỹ thuật khác (như Chia để trị). Với kỹ thuật chia để trị, không rõ ràng liệu kỹ thuật này là nhanh hay chậm. Lý do là ở mỗi mức của đệ quy kích thước nhỏ hơn và số lượng của bài toán con lớn hơn.
* Nếu đối với một bài toán cụ thể mà ta có thể chứng minh kỹ thuật Tham lam cho kết quả tối ưu toàn cục thì thuật toán này thường sẽ là một lựa chọn tốt nhất vì nó có thời gian thực thi nhanh hơn các phương pháp tối ưu khác như Quy hoạch động.

Nhược điểm:

* Khó khăn của tham lam là bạn rất vất vả để hiểu chính xác vấn đề. Thậm chí với giải thuật chính xác rồi, cũng rất khó khăn để chứng minh tại sao nó đúng. Chứng minh một giải thuật tham lam là đúng có cảm giác là cả một nghệ thuật hơn là một khoa học, vì nó đòi hỏi rất nhiều sự sáng tạo.
* Đa số các thuật toán áp dụng kỹ thuật Tham lam chỉ cho lời giải tương đối chính xác.

- Khó hiểu và khó để chứng minh một lời giải áp dụng kỹ thuật Tham lam là tối ưu toàn cục (nếu nó thực sự tối ưu toàn cục).

3.2 Ví dụ

3.2.1 Dijkstra’s Algorithm (Tìm đường đi ngắn nhất trong đồ thị có trọng số)

3.2.1.1 Ý tưởng thuật toán

Thuật toán Dijstra cho phép tìm đường đi ngắn nhất từ một đỉnh s đến các đỉnh còn lại của đồ thị và trọng số tương ứng.

Phương pháp của thuật toán là xác định tuần tự đỉnh có chiều dài đến s theo thứ tự tăng dần.

Thuật toán được xây dựng dựa trên cơ sở gán cho mỗi đỉnh các nhãn tạm thời. Nhãn tạm thời của các đỉnh cho biết cận trên của chiều dài đường đi ngắn nhất từ s đến đỉnh đó. Nhãn của các đỉnh sẽ biến đổi trong các bước lặp, mà mỗi bước lặp sẽ có một nhãn tạm thời trở thành chính thức. Nếu nhãn của một đỉnh nào đó trở thành chính thức thì nó cũng chính là chiều dài ngắn nhất từ đường đi s đến đỉnh đó.

3.2.1.2 Mô tả thuật toán

Ký hiệu:

* L(v) để chỉ nhãn của đỉnh, tức là cận trên của chiều dài đường đi ngắn nhất từ s0 đến v.
* d (s0, v) là chiều dài đường đi ngắn nhất từ s0 đến v.
* m (s0, v) là trọng số của cạnh (s, v).

Thuật toán Dijstra tìm chiều dài đường đi ngắn nhất từ đỉnh s đến n-1 đỉnh còn lại được mô tả như sau:

Input: G, s0

Output: d (s0, v), v s0

+ Khởi động:

L (v) = , // Nhãn tạm thời

S = ; // Tập lưu trữ các đỉnh có nhãn chính thức.

+ Bước 0:

d (s0, s0) = L (s0) = 0;

S = {s0}; // s0 có nhãn chính thức

+ Bước 1:

Tính lại nhãn tạm thời L (v), v S:

Nếu v kề với s0 thì: L (v) = Min {L(v), L(s0) + m (s0, v)};

Tìm s1 S và kề với s0 sao cho:

L (s1) = Min {L (v): v S};

(Khi đó: d (s0, s1) = L (s1))

S = S {s1}; // S = {s0, s1}; s1 có nhãn chính thức.

+ Bước 2:

Tính lại nhãn tạm thời L(v), v S:

Nếu v kề với s1 thì: L(v) = Min {L(v), L(s1) + m (s1, v)};

Tìm s2 S và kề với s1 hoặc s0 sao cho:

L(s2) = min {L(v): {v S};

(Khi đó: d (s, s2) = L(s2): // d (s0, s1) d (s0, s2)

Nếu L (s2) = Min {L (sj), L (sj) + m (sj, s2)} thì đường đi từ s đến s2 qua đỉnh sj là ngắn nhất, và sj là đỉnh đứng kề trước s2.

S = S {s2} // S = {s0, s1, s2}; s2 có nhãn chính thức.

…

+ Bước i:

Tính lại nhãn tạm thời L (v), v S:

Nếu v kề với si-1 thì: L(v) = Min {L(v), L(si-1) + m (si-1, v)};

Tìm si S và kề với sj, j = 0, i-1 sao cho:

L(si) = Min {L (v): v S};

(d (s, si) = L (si)); // 0 = d (s, s1) d (s, s2) … d (s, si)

Nếu L (si) = Min {L (sj), L (sj) + m (sj, si)} thì đường đi ngắn nhất từ s đến si đi qua đỉnh sj và sj là đỉnh đứng kề trước si.

S = S si // S = {s0, s1, s2, …, si}, si có nhãn chính thức.

Thuật toán dừng khi i = n-1;

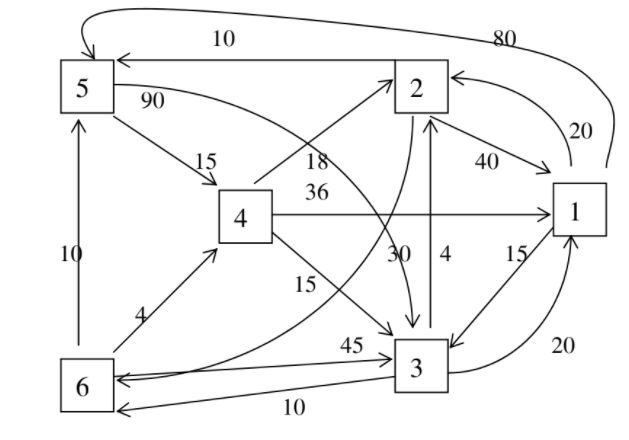
Khi thuật toán kết thúc, ta có:

0 = d (s0, s0) d (s0, s1) d (s0, s2) … d (s0, sn-1)

Nếu chỉ tìm đường đi ngắn nhất từ s0 đến t thì thuật toán dừng khi có t S.

Tính chất tham lam của thuật toán Dijstra là tại mỗi bước, chọn si S và si là đỉnh kề với sj, với j = 0, i-1 sao cho L (si) = Min {L (v): v S}.

Minh họa: xét đồ thị có hướng G sau:



Hình 3.1

Ta có kết quả tính toán sau đây:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bước lặp | Đường đi ngắn nhất là đường đi từ đỉnh 1 | Đến đỉnh | Chiều dài của đường đi ngắn nhất từ đỉnh s (=1) đến các đỉnh khác: | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Bước 1 | 1 3 | 3 | - | 20 | 15 |  | 80 |  |
| Bước 2 | 1 3 2 | 2 | - | 19 | - |  |  | 25 |
| Bước 3 | 1 3 6 | 6 | - | - | - |  | 29 | 25 |
| Bước 4 | 1 3 6 4 | 4 | - | - | - | 29 |  | - |
| Bước 5 | 1 3 2 5 | 5 | - | - | - | - | 29 | - |

Bảng 3.1

3.2.1.3 Triển khai bằng python 3

|  |
| --- |
| **import** sys  **class** **Graph**():  **def** \_\_**init**\_\_(self, vertices):  self.V = vertices  self.graph = [[0 for column in range(vertices)]  **for** row **in** range(vertices)]  **def** **printSolution**(self, dist):  **print**("Vertex tDistance from Source")  **for** node **in** range(self.V):  **print**(node, "t", dist[node])  # Hàm tìm đỉnh với giá trị khoảng cách tối thiểu từ  # tập hợp các đỉnh chưa được đưa vào cây đường đi ngắn nhất.  **def** **minDistance**(self, dist, sptSet):  # Bắt đầu khoảng cách ngắn nhất cho nút tiếp theo  min = sys.maxsize  # Tìm kiếm không phải đỉnh gần nhất cũng không phải cây ngắn nhất  **for** v **in** range(self.V):  **if** dist[v] < min **and** sptSet[v] == False:  min = dist[v]  min\_index = v  **return** min\_index  # hàm triển khai thuật toán Dijstra's sử dụng cách biểu diễn ma trận kề.  **def** **dijkstra**(self, src):  dist = [sys.maxsize] \* self.V  dist[src] = 0  sptSet = [False] \* self.V  **for** cout **in** range(self.V):  # Chọn đỉnh có khoảng cách ngắn nhất từ tập hợp đỉnh chưa được xử lý  # u luôn bằng src trong lần lặp đầu tiên  u = self.minDistance(dist, sptSet)  # Đặt khoảng cách ngắn nhất của đỉnh trong cây  sptSet[u] = True  # Cập nhật giá trị dist của các đỉnh liền kề của đỉnh được chọn  # chỉ khi khoảng cách hiện tại  # và khoảng cách mới không có trong cây đường đi ngắn nhất  **for** v **in** range(self.V):  **if** self.graph[u][v] > 0 and \  sptSet[v] == False and \  dist[v] > dist[u] + self.graph[u][v]:  dist[v] = dist[u] + self.graph[u][v]  self.printSolution(dist)  g = Graph(6)  g.graph = [[0,20,15,0,80,0],  [40,0,0,0,10,15],  [20,4,0,0,0,10],  [36,18,15,0,0,0],  [0,90,15,0,0,0],  [0,0,45,4,10,0]  ];  g.dijkstra(0); |

3.2.2 Prim’s Algorithm

3.2.2.1 Ý tưởng thuật toán

Thuật toán Prim xây dựng một đồ thị con T của G như sau:

Đầu tiên chọn tùy ý một đỉnh của G đặt vào T.

Quá trình sau còn thực hiện trong khi T chưa chứa hết các đỉnh của G:

Mỗi bước, tìm một cạnh có trọng số nhỏ nhất nối một đỉnh trong T với một đỉnh ngoài T. Thêm cạnh này vào T.

Kết thúc thuật toán Prim cho ta một MST của đồ thị G.

Tính tham lam của thuật toán Prim là tại mỗi bước thêm vào T một cạnh có trọng số nhỏ nhất nối một đỉnh trong T và một đỉnh ngoài T.

3.2.2.2 Mô tả thuật toán

Thuật toán xuất phát từ một cây chỉ chứa đúng một đỉnh và mở rộng từng bước một, mỗi bước thêm một cạnh mới vào cây, cho tới khi bao trùm được tất cả các đỉnh của đồ thị.

Input: Một đồ thị có trọng số liên thông với tập hợp đỉnh *V* và tập hợp cạnh *E* (trọng số có thể âm). Đồng thời cũng dùng *V* và *E* để ký hiệu số đỉnh và số cạnh của đồ thị.

Khởi tạo: *V*mới = {*x*}, trong đó *x* là một đỉnh bất kì (đỉnh bắt đầu) trong *V*, *E*mới ={}

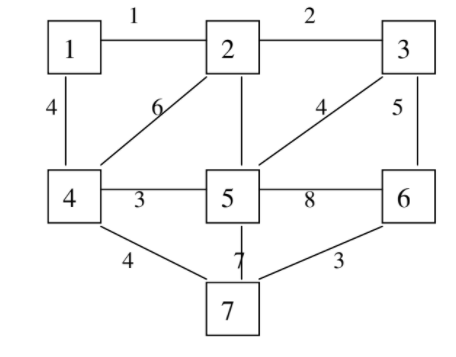
Lặp lại cho tới khi *V*mới = *V*:

Chọn cạnh (*u*, *v*) có trọng số nhỏ nhất thỏa mãn *u* thuộc *V*mới và *v* không thuộc *V*mới (nếu có nhiều cạnh như vậy thì chọn một cạnh bất kì trong chúng)

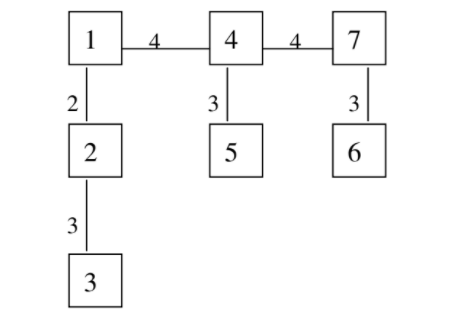
Thêm *v* vào *V*mới, và thêm cạnh (*u*, *v*) vào *E*mới

Output: *V*mới và *E*mới là tập hợp đỉnh và tập hợp cạnh của một cây bao trùm nhỏ nhất

Vd minh họa: Xét đồ thị sau:



Hình 3.2



Hình 3.3

|  |  |  |
| --- | --- | --- |
| Bước | Các cạnh chọn | Tập U |
| 0 | - | {1} |
| 1 | (2,1) | {1,2} |
| 2 | (3,2) | {1,2,3} |
| 3 | (4,1) | {1,2,3,4} |
| 4 | (5,4) | {1,2,3,4,5} |
| 5 | (7,4) | {1,2,3,4,5,7} |
| 6 | (6,7) | {1,2,3,4,5,7,6} |

Bảng 3.2

3.2.2.3 Triển khai bằng python 3

|  |
| --- |
| class Python():  def \_\_init\_\_(self, vertices):  self.V = vertices  self.graph = [[0 for column in range(vertices)]  for row in range(vertices)]  def printMST(self, parent):  print("Edge \tWeight")  for i in range(1, self.V):  print(parent[i], "-", i, "\t", self.graph[i][parent[i]])  def minKey(self, key, mstSet):  min = 1000000  for v in range(self.V):  if key[v] < min and mstSet[v] == False:  min = key[v]  min\_index = v  return min\_index  def primMST(self):  key = [1000000] \* self.V  parent = [None] \* self.V  key[0] = 0  mstSet = [False] \* self.V  parent[0] = -1  for cout in range(self.V):  u = self.minKey(key, mstSet)  mstSet[u] = True  for v in range(self.V):  if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:  key[v] = self.graph[u][v]  parent[v] = u  self.printMST(parent)  g = Python(7)   1. graph =[ [0,1,0,4,0,0,0],   [1,0,2,6,6,0,0],  [0,2,0,0,4,5,0],  [4,6,0,0,3,0,4],  [0,6,4,3,0,8,7],  [0,0,5,0,8,0,3],  [0,0,0,4,7,3,0] ]  g.primMST(); |

3.2.3 Huffman Trees and Codes

3.2.3.1 Ý tưởng thuật toán

Trong bản mã ASCII, mỗi ký tự được biểu diễn bằng chuối 8 bit.

Giảm số bit để biểu diễn 1 ký tự. Dùng chuỗi bit ngắn hơn để biểu diễn ký tự xuất hiện nhiều. Sử dụng mã tiền tố để phân cách các ký tự.

* B1: Duyệt file, lập bảng thống kê tần suất xuất hiện của mỗi ký tự.
* B2: Xây dựng cây Huffman dựa vào bảng thống kê.
* B3: Sinh mã Huffman cho mỗi ký tự dựa vào cây Huffman.
* B4: Duyệt file, thay toàn bộ ký tự bằng mã Huffman tương ứng.
* B5: Lưu lại cây Huffman (bảng mã) dùng cho việc giải nén, xuất file đã nén.

Ví dụ minh họa:

Chuỗi ký tự cần nén:

F=“ABABBCBBDEEEABABBAEEDDCCABBBCDEEDCBCCCCDBBBCAAA”, n = 47

Bảng tuần suất xuất hiện:

|  |  |
| --- | --- |
| Ký tự | Tần suất |
| A | 9 |
| B | 15 |
| C | 10 |
| D | 6 |
| E | 7 |

Bảng 3.3

Xây dựng cây Huffman với thuật toán tham lam:

Trong giải thuật tham lam giải bài toán xây dựng cây mã tiền tố tối ưu của

Huffman, ở mỗi bước ta chọn hai chữ cái có tần số thấp nhất để mã hóa bằng từ mã dài nhất. Giả sử có tập A gồm {\displaystyle n}ký hiệu và hàm trọng số tương ứng {\displaystyle W(i),i=1..n}.

* Khởi tạo: Tạo một rừng gồm n cây, mỗi cây chỉ có một nút gốc, mỗi nút gốc tương ứng với một ký tự và có trọng số là tần số/tần suất của ký tự đó {\displaystyle W(i)}.
* Lặp:

+ Mỗi bước sau thực hiện cho đến khi rừng chỉ còn một cây:

+ Chọn hai cây có trọng số ở gốc nhỏ nhất hợp thành một cây bằng cách thêm một gốc mới nối với hai gốc đã chọn. Trọng số của gốc mới bằng tổng trọng số của hai gốc tạo thành nó.

Như vậy ở mỗi bước số cây bớt đi một. Khi rừng chỉ còn một cây thì cây đó biểu diễn mã tiền tố tối ưu với các ký tự đặt ở các lá tương ứng.

3.2.3.2 Mô tả thuật toán

Input: Bảng chữ cái A = {a1, a2, a3, …, an}

Tập các trọng số (tần suất xuất hiện tương ứng W = {w1, w2, …, wn}

Ví dụ: wi = weight(ai), 1 i n.

Output: Bộ mã C (A, W) = {c1, c2, ..., cn} là tập hợp các từ mã (nhị phân) trong đó ci là từ mã ai, 1.i n.

3.2.3.3 Triển khai bằng python 3

|  |
| --- |
| # Huffman Coding in python  # file = open('1955852.html','r',encoding='utf-8')  # string = file.read()  #string = 'Nguyen Thi Huyen Trang'  # Creating tree nodes  **class** NodeTree(object):  **def** \_\_init\_\_(self, left=None, right=None):  self.left = left  self.right = right  **def** children(self):  **return** (self.left, self.right)  **def** nodes(self):  **return** (self.left, self.right)  **def** \_\_str\_\_(self):  **return** '%s\_%s' % (self.left, self.right)  # Main function implementing huffman coding  **def** huffman\_code\_tree(node, left=True, binString=''):  **if** type(node) is str:  **return** {node: binString}  (l, r) = node.children()  d = dict()  d.update(huffman\_code\_tree(l, True, binString + '0'))  d.update(huffman\_code\_tree(r, False, binString + '1'))  **return** d  # Calculating frequency  freq = {}  **for** c **in** string:  **if** c **in** freq:  freq[c] += 1  **else**:  freq[c] = 1  freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)  nodes = freq  **while** len(nodes) > 1:  (key1, c1) = nodes[-1]  (key2, c2) = nodes[-2]  nodes = nodes[:-2]  node = NodeTree(key1, key2)  nodes.append((node, c1 + c2))  nodes = sorted(nodes, key=lambda x: x[1], reverse=True)  huffmanCode = huffman\_code\_tree(nodes[0][0])  **print**(' Char | Huffman code ')  **print**('----------------------')  **for** (char, frequency) in freq:  **print**(' %-4r |%12s' % (char, huffmanCode[char]))    #Encode-----------------------------------------------  file1 = open('encode.txt','w+')  **for** c **in** string:  file1.write(huffmanCode[c])  file1.write(' ')  file1.close()  #Decode  encode = open('encode.txt', 'r', encoding='utf-8')  encode\_string = encode.read()  huffmanDeCode = dict()  **for** i **in** huffmanCode:  huffmanDeCode[huffmanCode[i]] = i  #print(type(encode\_string))  encode\_string = encode\_string.split()  #print(encode\_string)  decode = open('decode.txt','w+',encoding='utf-8')  **for** c **in** encode\_string:  **if** c is not '':  decode.write(huffmanDeCode[c]) |

CHƯƠNG 4 – DYNAMIC PROGRAMMING

* 1. Giới thiệu

Quy hoạch động (DP – Dynamic Programming) là một thuật ngữ được nhà toán học Rechard Bellman đưa ra vào năm 1957. Một phương pháp giải bài toán bằng cách kết hợp các lời giải cho các bài toán con của nó giống như phương pháp chia để trị (Devide and conquer). Các thuật toán chia để trị phân hoạch bài toán càn giải quyết thành các bài toán con độc lập với nhau, sau đó giải quyết chúng bằng phương pháp đệ quy (recursive) và kết hợp các lời giải lại để tìm được lời giải cho bài toán ban đầu. Ngược lại quy hoạch động là phương pháp được áp dụng khi mà các bài toán con của bài toán ban đầu là không độc lập với nhau – chúng có chung các bài toán con nhỏ hơn. Trong các trường hợp như vậy một thuật toán chia để trị sẽ thực hiện nhiều việc hơn những gì thực sự cần thiết, nó sẽ lặp lại việc giải quyết các bài toán con nhỏ một lần duy nhất sau đó lưu kết quả vào một bảng. Điều nà giúp nó tránh không phải tính toán lại các kết quả mỗi khi gặp một bài toán con nhỏ nào đó.

Quy hoạch động thường được áp dụng với các bài toán tối ưu. Trong các bài toán tối ưu đó thương có nhiều lời giải. mỗi lời giải có một giá trị được lượn giá bằng cách sử dụng một hàm đánh giá tùy thuộc vào các bài toán cụ thể và yêu cầu của bài toán cụ thể và yêu cầu của bài toán là tìm ra một nghiệm có giá trị của hàm đánh giá là tối ưu (lớn nhất hoặc nhỏ nhất).

Quy hoạch động là phương pháp chung rất hiệu quả để giải quyết các vấn đề tối ưu chẳng hạn như trên các đối tượng sắp thứ tự từ trái qua phải, tìm đường đi ngắn nhất, vấn đề điều khiển tối ưu, …

4.2 Ví dụ

4.2.1 Coin-row problem (Vấn đề về hàng đồng xu)

4.2.1.1 Ý tưởng thuật toán

Gọi F (n) là số tiền lớn nhất có thể nhặt được từ hàng n đồng xu. Để tính toán một lần lặp lại cho F (n), chúng ta phân vùng tất cả các đồng xu được phép lựa chọn thành hai nhóm: những nhóm bao gồm đồng xu cuối cùng và những nhóm không có nó.

Số tiền lớn nhất chúng ta có thể nhận được từ nhóm đầu tiên bằng

cn + F (n - 2) —giá trị của đồng xu thứ n cộng với số tiền tối đa chúng ta có thể nhặt được từ n - 2 xu đầu tiên. Các số tiền tối đa chúng ta có thể nhận được từ nhóm thứ hai bằng F (n - 1) định nghĩa bởi F (n). Do đó, chúng ta có sự lặp lại sau đây là đối tượng rõ ràng điều kiện ban đầu:

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

F (0) = 0, F (1) = c1. (8,3)

Chúng ta có thể tính F (n) bằng cách điền vào bảng một hàng từ trái sang phải theo cách tương tự như cách nó được thực hiện đối với số Fibonacci thứ n bởi Thuật toán Fib (n).

4.2.1.2 Mô tả thuật toán

|  |
| --- |
| //Input: Mảng C [1, 2, …, n] gồm các số nguyên dương cho biết giá trị tiền xu.  //Output: Số tiền tối đa có thể nhặt được.  F[0]← 0  F[1]← C[1]  for i ← 2 to n do  F[i]← max(C[i] + F[i − 2], F[i − 1])  return F[n] |

4.2.1.3 Triển khai bằng python 3

|  |
| --- |
| **def** **coinRow**(C):  F = [0] \* (n+1)  F[0] = 0  F[1] = C[1]  **for** i **in** range(2, n+1):  F[i] = max(C[i] + F[i-2], F[i-1])  **return** F[n] |

4.2.2 Change-making problem (Vấn đề tạo thay đổi)

4.2.2.1 Ý tưởng thuật toán

Trao tiền lẻ cho số tiền n sử dụng mức tối thiểu số lượng tiền có mệnh giá d1 < d2 <... < dm. Đối với mệnh giá tiền xu được sử dụng ở Hoa Kỳ, cũng như những thứ được sử dụng ở hầu hết nếu không phải tất cả các quốc gia khác, có một thuật toán rất đơn giản và hiệu quả được thảo luận dưới đây. Đây, chúng tôi xem xét một thuật toán lập trình động cho trường hợp chung, giả sử có sẵn số lượng không giới hạn tiền xu cho mỗi mệnh giá m d1 <d2 <... <dm trong đó d1 = 1.

Gọi F (n) là số đồng xu tối thiểu mà giá trị của chúng lên đến n; nó là

thuận tiện để xác định F (0) = 0. Số lượng n chỉ có thể nhận được bằng cách thêm một đồng xu có mệnh giá dj thành số tiền n - dj với

j = 1, 2, ..., m sao cho n ≥ dj.

Do đó, chúng tôi có thể xem xét tất cả các mệnh giá như vậy và chọn một mệnh giá tối thiểu F (n - dj) + 1. Vì 1 là hằng số nên tất nhiên chúng ta có thể tìm F (n - dj) nhỏ nhất đầu tiên và sau đó thêm 1 vào nó. Do đó, chúng ta có sự lặp lại sau cho F (n):

F (n) = min F (n - dj)} + 1 for n > 0,

j: n≥dj

F (0) = 0.

Chúng ta có thể tính F (n) bằng cách điền vào bảng một hàng từ trái sang phải theo cách tương tự theo cách nó đã được thực hiện ở trên cho vấn đề hàng xu, nhưng tính toán một bảng mục nhập ở đây yêu cầu tìm số tối thiểu lên đến m số.

4.2.2.2 Mô tả thuật toán

|  |
| --- |
| //Input: Số nguyên dương n và mảng số nguyên dương D[1,2… m] tăng dần.  //Output: Số xu tối thiểu cộng vào n  F[0]← 0  For i ← 1 to n do  temp ← ∞; j ← 1  while j ≤ m and i ≥ D[j ] do  temp ← min(F[i − D[j ]], temp)  j ← j + 1  F[i]← temp + 1  return F[n] |

Độ phức tạp : Thời gian và không gian hiệu quả của thuật toán rõ ràng là O (nm) và (n), tương ứng.

4.2.2.3 Triển khai bằng python 3

|  |
| --- |
| INF = 99999999  n = 6  m = 2  D = [1,3,4]  **def** **changeMaking**(D, n):  F = [0]\* (n+1)  F[0] = 0  **for** i **in** range(1, n+1):  temp = INF  j = 1  **while** (j <= m) **and** (i >= D[j]):  temp = min(F[i-D[j]], temp)  j = j + 1  F[i] = temp + 1  **return** F[n]  **print**(**changeMaking**(D, n)) |

4.2.3 Coin-collecting problem (Vấn đề thu thập tiền xu)

4.2.3.1 Ý tưởng thuật toán

Gọi F (i, j) là số đồng xu lớn nhất mà robot có thể thu thập và mang đến ô (i, j) ở hàng thứ i và cột thứ j của bàn cờ. Nó có thể đến ô này từ ô liền kề (i - 1, j) phía trên nó hoặc từ ô liền kề (i, j - 1) ở bên trái của nó. Số đồng xu lớn nhất có thể được mang đến các ô này lần lượt là F (i - 1, j) và F (i, j - 1). Tất nhiên, không có ô liền kề nào phía trên các ô trong hàng đầu tiên và không có ô liền kề nào ở bên trái các ô trong cột đầu tiên. Đối với các ô đó, chúng ta giả sử rằng F (i - 1, j) và F (i, j - 1) bằng 0 đối với các ô lân cận không tồn tại của chúng. Do đó, số xu lớn nhất mà rô bốt có thể mang đến ô (i, j) là giá trị lớn nhất của hai số này cộng với một đồng xu có thể có tại chính ô (i, j). Nói cách khác, chúng ta có công thức sau cho F (i, j):

F (i, j) = max {F (i - 1, j), F (i, j - 1)} + cij for 1 ≤ i ≤ n, 1 ≤ j ≤ m

F (0, j) = 0 cho 1 ≤ j ≤ m và F (i, 0) = 0 for 1 ≤ i ≤ n,

trong đó cij = 1 nếu có đồng xu trong ô (i, j) và cij = 0 nếu không.

Sử dụng các công thức này, chúng ta có thể điền vào bảng n × m các giá trị F (i, j) một trong hai hàng

theo hàng hoặc từng cột, như điển hình cho các thuật toán lập trình động

liên quan đến các bảng hai chiều.

4.2.3.2 Mô tả thuật toán

Input: Ma trận C [1...n, 1…m] có các phần tử bằng 1 và 0 cho các ô không có đồng xu tương ứng.

Output: Số lượng đồng xu lớn nhất mà robot có thể mang đến ô (n, m)

Phân tích thuật toán

Truy tìm lại đường đi tối ưu:

* Có thể dò ngược các phép tính để có được đường đi tối ưu.
* Nếu F (i-1, j)> F (i, j-1), một đường dẫn tối ưu đến ô (i, j) phải đi xuống từ ô liền kề phía trên nó;
* Nếu F (i-1, j) <F (i, j-1), một đường dẫn tối ưu đến ô (i, j) phải đến từ ô liền kề bên trái;
* Nếu F (i-1, j) = F (i, j-1), nó có thể đến ô (i, j) từ một trong hai hướng. Có thể bỏ qua ràng buộc bằng cách ưu tiên đến từ ô liền kề bên trái.
* Nếu chỉ có một lựa chọn, tức là F (i-1, j) hoặc F (i, j-1) không khả dụng, hãy sử dụng lựa chọn có sẵn khác.

- Con đường tối ưu có thể đạt được trong thời gian Θ (n + m).

4.2.3.3 Triển khai bằng python 3

|  |
| --- |
| n = 3 #row  m = 3 #colum  F = [[0]\*(n+1) for i in range(m+1)]  C = [[1,0,1],  [1,0,1],  [1,1,1]]  def RobotCoinCollection(C):  F[1][1] = C[1][1]  for j in range(2,m+1):  F[1][j] = F[1][j-1] + C[1][j]  for i in range(2,n+1):  F[i][1] = F[i-1][1] + C[i][1]  for j in range(2,m+1):  F[i][j] = max(F[i-1][j], F[i][j-1] + C[i][j])  return F[n][m]  print(RobotCoinCollection(C)) |

CHƯƠNG 5 – TRANSFORM-AND-CONQUER

Chương này đề cập đến một nhóm các phương pháp thiết kế dựa trên ý tưởng chuyển đổi. Chúng tôi gọi kỹ thuật chung này là biến đổi để trị bởi vì các phương pháp này hoạt động như các thủ tục hai giai đoạn. Đầu tiên, trong giai đoạn chuyển đổi, trường hợp của vấn đề được sửa đổi để dễ giải quyết hơn. Sau đó, trong giai đoạn thứ hai hoặc giai đoạn “trị”, vấn đề được giải quyết.

Có 3 biến thể chính của ý tưởng này và chúng khác nhau bởi những gì ta biến đổi yêu cầu đã cho:

* Chuyển đổi sáng một yêu cầu đơn giản hơn hoặc thuận tiện hơn của cùng 1 vấn đề - dơn giản hóa yêu cầu(đơn giản hóa yêu cầu - instance simplification)
* Chuyển đổi sang một biểu diễn khác của cùng bài toán(biến đổi biểu diễn - representation change)
* Chuyển đổi đưa đến một thể hiện khác của một bài toán khác đã có tồn tại giả thuật (thu giảm bài toán - problem reduction)

5.1 Presorting (Sắp xếp trước)

5.1.1 Ý tưởng thuật toán

Phần này đề cập đến ý tưởng đơn giản nhưng hiệu quả của việc sắp xếp trước. Nhiều bài toán thuật toán dễ giải hơn nếu đầu vào của chúng được sắp xếp.Sắp xếp trước là một ý tưởng cũ trong khoa học máy tính. Rõ ràng, hiệu quả về thời gian của các thuật toán liên quan đến sắp xếp có thể phụ thuộc vào hiệu quả của thuật toán sắp xếp đang được sử dụng.

Để đơn giản, giả định rằng danh sách được triển khai dưới dạng mảng, vì một số thuật toán sắp xếp dễ triển khai hơn đối với biểu diễn mảng.Sắp xếp trước "là một ví dụ phổ biến của" đơn giản hóa trường hợp ". Sắp xếp trước là sắp xếp trước thời gian, để tạo ra các giải pháp lặp lại nhanh hơn.

Ví Dụ: Kiểm tra tính duy nhất của phần tử trong một mảng.Ở ví dụ này ta sẽ xem xét một thuật toán brute-force cho vấn đề so sánh các cặp phần tử của mảng cho đến khi hai phần tử bằng nhau không còn hoặc không còn cặp nào cả. Hiệu suất trong trường hợp xấu nhất của thuật toán là θ()

Ngoài ra, chúng ta có thể sắp xếp mảng trước và sau đó chỉ kiểm tra các phần tử liên tiếp của nó: nếu mảng có các phần tử bằng nhau, thì một cặp trong số chúng phải cạnh nhau và ngược lại.

5.1.2 Mô tả thuật toán

|  |
| --- |
| // Giải quyết vấn đề về tính duy nhất của phần tử bằng cách sắp xếp mảng trước  // Đầu vào: Một mảng A [0..n - 1] gồm các phần tử có thể sắp xếp thứ tự  // Kết quả: Trả về "true" nếu A không có phần tử nào bằng nhau, "false" nếu không  **for** *i* ← 0 **to** *n* − 2 **do**  **if** *A*[*i*] = *A*[*i* + 1]  **return false**  **return true** |

Thời gian chạy của thuật toán này là tổng thời gian dành cho việc sắp xếp và thời gian dành cho việc kiểm tra các phần tử liên tiếp. Vì phần trước yêu cầu ít nhất n log n phép so sánh và phần sau cần không quá n-1 phép so sánh, nên phần sắp xếp sẽ xác định hiệu quả tổng thể của thuật toán. Vì vậy, nếu chúng ta sử dụng thuật toán sắp xếp bậc hai ở đây, toàn bộ thuật toán sẽ không hiệu quả hơn thuật toán brute-force. Nhưng nếu chúng ta sử dụng một thuật toán sắp xếp tốt, chẳng hạn như merge sort, với hiệu suất trong trường hợp xấu nhất là (n log n), thì hiệu quả trong trường hợp xấu nhất của toàn bộ thuật toán dựa trên sắp xếp trước cũng sẽ bằng (n log n):

T (n) = Tsort(n) + Tscan(n) ∈θ(n log n) +θ(n) = θ(n log n).

5.1.3 Triển khai bằng python 3

|  |
| --- |
| **def** Isequal(A):  **for** i in range(len(A)-1):  **if** A[i] == A[i+1]:  **return** False  **return** True |

*5.2* Gaussian Elimination (Phép loại trừ Gaus)

5.2.1 Ý tưởng thuật toán

Cho hệ phương trình gồm n phương trình với n ẩn số



Để giải hệ phương trình trên, ta dùng giải thuật loại trừ Gauss (Gauss elimination)

Ý tưởng: biến đổi hệ thống n phương trình tuyến tính với *n* biến thành một hệ thống tương đương (tức là có cùng lời giải như hệ phương trình ban đầu) với ma trận tam giác trên (một ma trận với các hệ số zero dưới đường chéo chính)

Giải thuật Gauss thể hiện tinh thần của chiến lược biến thể - để - trị theo kiểu “đơn giản hóa thể hiện” (instance simplification).

 => 

Làm cách nào để ta có thể chuyển một hệ thống với ma trận A bất kỳ thành một hệ thống trương đương với ma trận tam giác trên A’?

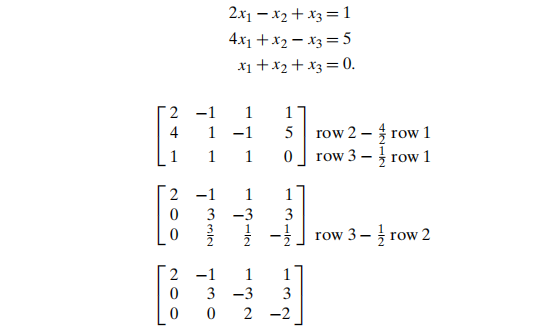
Bằng một loạt các phép biến đổi cơ bản như sau:

- Hoán vị hai phương trình trong hệ thống

- Thay một phương trình bằng phương trình đó nhân với một hệ số

- Thay một phương trình với tổng hay hiệu phương trình đó với một phương trình khác được nhân một hệ số

Ví dụ : Giải hệ phương trình bằng thuật toán loại trừ Gauss



Chúng ta có thể giải quyết bằng cách thay thế ngược lại

,, 

5.2.2 Mô tả thuật toán

|  |
| --- |
| //ForwardElimination(A[1..n,1..n],b[1..n])  // Áp dụng loại bỏ Gaussian cho ma trận A của các hệ số của hệ thống,  // được tăng cường với vectơ b của các giá trị bên phải của hệ thống  // Đầu vào: Ma trận A [1..n, 1..n] và vectơ cột b [1..n]  // Đầu ra: Một ma trận tam giác trên tương đương thay cho A với  // các giá trị bên phải tương ứng trong cột st (n + 1)  **for** *i* ← 1 **to** *n* **do** *A*[*i, n* + 1]← *b*[*i*]  **for** *i* ← 1 **to** *n* − 1 **do**  **for** *j* ← *i* + 1 **to** *n* **do**  **for** *k* ← *i* **to** *n* + 1 **do**  A[j, k]←A[j, k]−A[i, k] \*A[j, i]/A[i,i] |

5.2.3 Triển khai bằng python 3

|  |
| --- |
| **def** ForwardElimination(A,b):  n = len(A)  **for** i **in** range(n-1):  A[i][n+1]=b[i]  **for** i **in range(**n**):**  **for** j **in range(**i+1,n**):**  **for** k **in range(**i,n+1**):**  A[j][k] = A[j][k]-A[I][k]\*A[j][i]/A[i][i] |

5.3 Balanced Search Trees (Cây tìm kiếm cân bằng)

5.3.1 Ý tưởng thuật toán

Chúng ta đã từng thảo luận về cây tìm kiếm nhị phân. Nó là một cây nhị phân có các nút chứa các phần tử của một tập hợp các mục có thể xác định được, một phần tử trên mỗi nút, sao cho tất cả các phần tử trong cây con bên trái đều nhỏ hơn phần tử trong gốc của cây con và tất cả các phần tử trong cây con bên phải là lớn hơn nó.

Các nhà khoa học máy tính đã dành rất nhiều nỗ lực trong việc cố gắng tìm ra một cấu trúc duy trì các đặc tính tốt của cây tìm kiếm nhị phân cổ điển. Họ đã đưa ra hai cách tiếp cận.

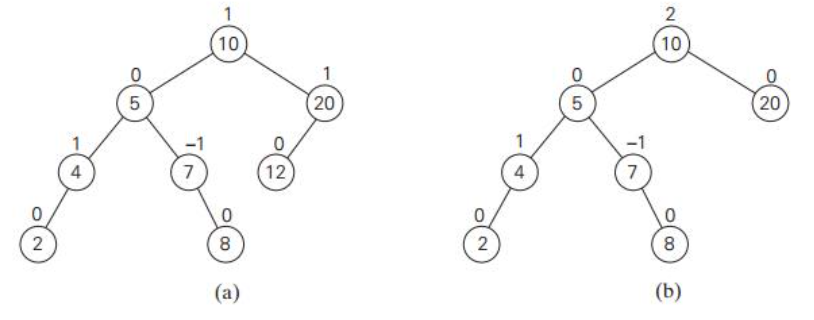
Cách tiếp cận đầu tiên là của sự đa dạng đơn giản hóa ví dụ: cây tìm kiếm nhị phân không cân bằng được chuyển đổi thành cây cân bằng. Bởi vì điều này, những cây như vậy được gọi là tự cân bằng. Các cách triển khai cụ thể của ý tưởng này khác nhau theo định nghĩa của chúng về sự cân bằng. Một cây AVL yêu cầu sự khác biệt giữa chiều cao của cây con bên trái và bên phải của mọi nút không bao giờ vượt quá 1. Một cây màu đỏ-đen cho phép chiều cao của một cây con lớn gấp đôi chiều cao của cây con kia của cùng một nút. Nếu việc chèn hoặc xóa một nút mới sẽ tạo ra một cây có yêu cầu về số dư bị vi phạm, cây được cấu trúc lại bởi một số các phép biến đổi đặc biệt được gọi là phép quay khôi phục số dư cần thiết. Trong phần này, chúng ta sẽ chỉ thảo luận về cây AVL.

Cách tiếp cận thứ hai là về sự đa dạng thay đổi biểu diễn: cho phép nhiều hơn một phần tử trong một nút của cây tìm kiếm. Những trường hợp cụ thể của những cây đó là cây 2-3, cây 2-3-4. Trong phần này, chúng ta sẽ cùng nhau thảo luận về câu 2-3.

* *Cây AVL*

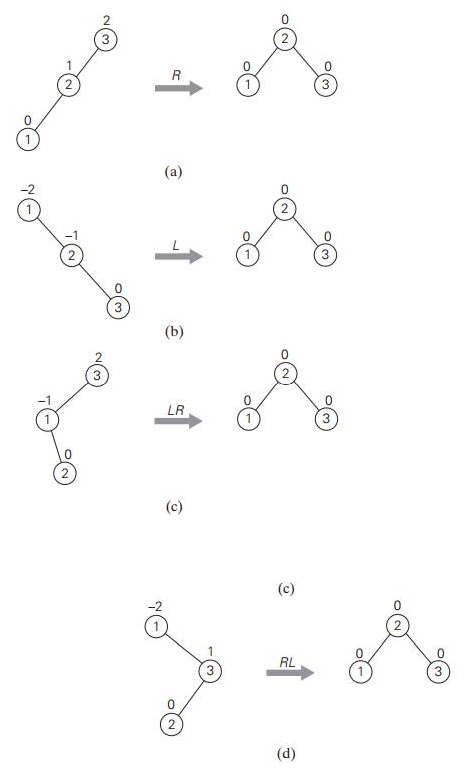
Định nghĩa: Một cây AVL là một cây tìm kiếm nhị phân, trong đó yếu tố cân bằng của mỗi nút, được định nghĩa là sự chênh lệch giữa độ cao của các cây con bên trái và bên phải của nút, là 0 hoặc +1 hoặc - 1. (Chiều cao của cây trống được định nghĩa là - 1. Tất nhiên, hệ số cân bằng cũng có thể được tính là hiệu số giữa các số cấp chứ không phải là chênh lệch độ cao của các cây con bên trái và bên phải của nút.)

Ví dụ, cây tìm kiếm nhị phân trong Hình 6.2a là cây AVL nhưng cây trong Hình 6.2b thì không.



Hình 5.1

Nếu việc chèn một nút mới làm cho cây AVL không cân bằng, chúng ta biến đổi cây bằng một phép quay. Một phép quay trong cây AVL là một phép biến đổi cục bộ của cây con bắt nguồn từ một nút có số dư trở thành +2 hoặc −2. Nếu có một số nút như vậy, chúng ta xoay cây gốc tại nút không cân bằng là nút gần nhất với lá mới được chèn. Bốn phép quay: single R-rotation, single L-rotation, double LR-rotation và double RL-rotation được thể hiện trong dưới đây



Hình 5.2

* *Cây 2-3*

Như đã đề cập ở trên, ý tưởng thứ hai để cân bằng cây tìm kiếm là cho phép nhiều

hơn một khóa trong cùng một nút của cây. Cách triển khai đơn giản nhất của ý tưởng này là cây 2-3. Cây 2-3 là một cây có thể có hai loại nút: nút 2 và nút 3. Một nút 2 chứa một khóa K duy nhất và có hai nút con: nút con bên trái đóng vai trò là gốc của cây con có khóa nhỏ hơn K và nút con bên phải đóng vai trò là gốc của cây con có khóa lớn hơn K. (Trong nói cách khác, nút 2 là cùng loại nút mà chúng ta có trong cây tìm kiếm nhị phân cổ điển.) Một nút 3 chứa hai khóa có thứ tự K1 và K2 (K1 <K2) và có ba khóa con, khóa con ngoài cùng bên trái đóng vai trò là gốc của cây con có khóa nhỏ hơn K1, ở giữa đóng vai trò là gốc của cây con có khóa giữa K1 và K2, và ngoài cùng bên phải đóng vai trò là gốc của cây con có khóa lớn hơn K2.

Yêu cầu cuối cùng của cây 2-3 là tất cả các lá của nó phải trên cùng một mức. Nói cách khác, cây 2-3 luôn cân bằng hoàn toàn về chiều cao: chiều dài đường đi từ gốc đến lá là như nhau đối với mọi lá. Đó là thuộc tính này có được bằng cách cho phép nhiều hơn một khóa trong cùng một nút của cây tìm kiếm.

Tìm kiếm một khóa K nhất định trong cây 2-3 khá đơn giản. Chúng ta bắt đầu từ gốc, nếu gốc là một nút 2, chúng ta làm như thể nó là một cây tìm kiếm nhị phân: chúng ta dừng lại nếu *K* bằng khóa của gốc hoặc tiếp tục tìm kiếm ở bên trái hoặc bên phải cây con nếu K tương ứng nhỏ hơn hoặc lớn hơn khóa của gốc. Nếu nút gốc là nút 3, chúng ta biết sau không quá hai lần so sánh khóa liệu có thể dừng tìm kiếm (nếu K bằng một trong các khóa của nút gốc) hay cần tiếp tục tiếp tục tìm kiếm ở cây nào trong ba cây con của nút gốc.

Chèn khóa mới trong cây 2-3 được thực hiện như sau. Trước hết, chúng ta luôn chèn một khóa K mới vào một lá, ngoại trừ cây trống. Lá thích hợp được tìm thấy bằng cách thực hiện tìm kiếm K. Nếu lá được đề cập là 2 nút, ta chèn K vào đó làm khóa đầu tiên hoặc khóa thứ hai, tùy thuộc vào việc K nhỏ hơn hay lớn hơn khóa cũ của nút. Nếu chiếc lá là 3 nút, chúng ta chia chiếc lá ra làm hai: chiếc lá nhỏ nhất trong ba chiếc chìa khóa (hai chiếc cũ và chiếc chìa mới) được đặt vào chiếc lá đầu tiên, chiếc chìa khóa lớn nhất được đặt vào chiếc lá thứ hai, và khóa giữa được thăng cấp thành cha mẹ của lá già. (Nếu lá là gốc của cây, một gốc mới sẽ được tạo ra để chấp nhận khóa giữa.) Lưu ý rằng việc thăng hạng khóa giữa cho khóa cha của nó có thể gây tràn khóa cha (nếu đó là nút 3 nút) và do đó có thể dẫn đến một số nút tách dọc theo chuỗi tổ tiên của lá.

5.3.2 Mô tả thuật toán

Ta định nghĩa một hàm đệ quy  **is\_balanced**() lấy nút gốc làm đối số và trả về giá trị boolean đại diện cho việc cây có cân bằng hay không.

Ta cũng hãy định nghĩa một hàm trợ giúp **get\_height**() trả về chiều cao của cây. Lưu ý rằng  **get\_height**() cũng được triển khai đệ quy

Hàm **is\_balanced**() trả về true nếu cây con bên phải và cây con bên trái được cân bằng và nếu chênh lệch giữa chiều cao của chúng không vượt quá 1.

5.3.3 Triển khai bằng python 3

|  |
| --- |
| **def** **get**\_**height**(root):  **if** root is None:  **return** 0  **return** 1 + max(get\_height(root.left)\      , **get**\_**height**(root.right))  **def** **is**\_**balanced**(root):      # a None tree is balanced  **if** root is None:  **return** True  **return** is\_balanced(root.right) and \      is\_balanced(root.left) and \      abs(get\_height(root.left) - get\_height(root.right)) <= 1 |

5.4 Heaps and Heapsort

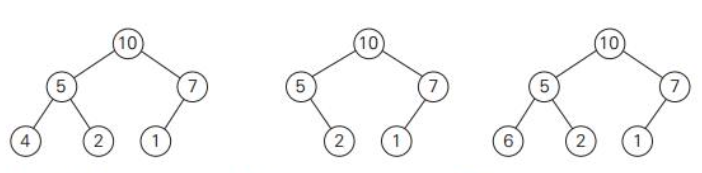
5.4.1 Ý tưởng thuật toán

Định nghĩa: Một heap có thể được định nghĩa là một cây nhị phân với các khóa được gán cho các nút của nó, một khóa cho mỗi nút, miễn là đáp ứng hai điều kiện sau:

Thuộc tính hình dạng: cây nhị phân về cơ bản là hoàn chỉnh (hoặc đơn giản là hoàn thành), tức là tất cả các cấp của nó đều đầy đủ ngoại trừ có thể là cấp cuối cùng, trong đó chỉ một số lá ngoài cùng bên phải có thể bị thiếu.

Thuộc tính đống: khóa trong mỗi nút lớn hơn hoặc bằng các khóa trong nút con của nó. (Điều kiện này được coi là tự động thỏa mãn cho tất cả các lá.)

Ví dụ, hãy xem các cây ở hình bên dưới. Cây đầu tiên là heap. Cái thứ hai không phải là heap, vì thuộc tính hình dạng của cây bị vi phạm. Và nút thứ ba không phải là heap, bởi vì sự thống trị không thành công đối với nút có khóa 5.



Hình 5.3

5.4.2 Mô tả thuật toán

|  |
| --- |
| //Thuật toán HeapBottomUp(H[1..n])  //Tạo một đống từ các phần tử của một mảng đã cho bằng thuật toán từ dưới lên  //Input: Mảng H[1..n] gồm các mục có thể sắp xếp được  //Output: Một heap H[1..n]  for i ← downto 1 do  k ← i; v ← H[k]  heap ← false  while not heap and 2 ∗ k ≤ n do  j ← 2 ∗ k  if j < n  if j< H[j + 1] j ← j + 1  if v ≥ H[j ]  heap ← true  else H[k]← H[j ]; k ← j  H[k]← v |

5.4.3 Triển khai bằng python 3

|  |
| --- |
| def HeapBottomUp(H):  n = len(H)  for i in range (int(n/2),0,-1):  k = i  v = H[k]  heap = False  while not heap and (2\*k)<=n:  j = 2\*k  if j < n:  if j < H[j+1]:  j = j + 1  if v >= H[j]:  heap = True  elif H[k] == H[j]:  k = j  H[k] = v  H = [2, 5, 4, 1, 9, 8]  HeapBottomUp(H)  n = len(H)  for i in range(n):  print ("%d" %H[i]), |

5.5 Horner’s Rule and Binary Exponentiation

5.5.1 Ý tưởng thuật toán

Trong phần này, chúng ta thảo luận về vấn đề tính toán giá trị của một đa thức

p (x) = anxn + an − 1xn − 1 + ... + a1x + a0 (6.1)

tại một điểm x cho trước và trường hợp đặc biệt quan trọng của tính toán xn. Đa thức tạo thành lớp chức năng quan trọng nhất vì chúng sở hữu rất nhiều một mặt thuộc tính tốt và có thể được sử dụng để ước lượng các loại các chức năng khác.

Bài toán sử dụng đa thức một cách hiệu quả có quan trọng trong vài thế kỷ; những khám phá mới vẫn đang được thực hiện 50 năm qua. Cho đến nay, điều quan trọng nhất trong số đó là **fast Fourier transform (FFT).** Tầm quan trọng thực tế của thuật toán đáng chú ý này, dựa trên biểu diễn một đa thức bằng các giá trị của nó tại các điểm được chọn đặc biệt, sao cho một số người coi đây là một trong những khám phá thuật toán quan trọng nhất thời gian.

* *Quy tắc Horner*

Quy tắc của Horner là một thuật toán cũ nhưng rất thanh lịch và hiệu quả để đánh giá đa thức. Nó được đặt theo tên của nhà toán học người Anh W. G. Horner, người đã xuất bản nó vào đầu thế kỷ 19. Nhưng theo Knuth, phương pháp được Isaac Newton sử dụng 150 năm trước Horner.

Quy tắc của Horner là một ví dụ điển hình về kỹ thuật thay đổi biểu diễn vì

nó dựa trên việc biểu diễn p (x) bằng một công thức khác với (6.1). Công thức mới này nhận được từ (6.1) bằng cách liên tiếp lấy x làm nhân tử chung trong phần còn lại đa thức bậc giảm dần:

p(x) = (... (anx + an−1)x + ...)x + a0. (6.2)

Ví dụ :

p(x) = 2x4 − x3 + 3x2 + x − 5, ta có:

p(x) = 2x4 − x3 + 3x2 + x − 5

= x(2x3 − x2 + 3x + 1) – 5

= x(x(2x2 − x + 3) + 1) − 5

= x(x(x(2x − 1) + 3) + 1) – 5

Trong công thức (6.2), chúng ta sẽ thay thế một giá trị của x tại đó đa thức cần được đánh giá. Như chúng ta sẽ thấy, không cần phải thực hiện rõ ràng quá trình chuyển đổi dẫn đến điều đó: tất cả những gì chúng ta cần là một danh sách ban đầu các hệ số của đa thức.

Việc tính toán bằng pen-and-pencil có thể được sắp xếp thuận tiện với một bảng hai nét. Hàng đầu tiên chứa các hệ số của đa thức (bao gồm tất cả các các hệ số bằng 0, nếu có) được liệt kê từ a0 cao nhất đến a0 thấp nhất.

Ngoại trừ cho mục nhập đầu tiên của nó, là một, hàng thứ hai được điền từ trái sang phải như sau: mục nhập tiếp theo được tính bằng giá trị của x nhân với mục nhập cuối cùng trong hàng thứ hai cộng với hệ số tiếp theo từ hàng đầu tiên. Mục nhập cuối cùng được tính theo kiểu này là giá trị đang được tìm kiếm.

* *Ví dụ:*

Đánh giá: e p(x) = 2x4 − x3 + 3x2 + x − 5 at x = 3.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| coefficients | 2 | -1 | 3 | 1 | -5 |
| x = 3 | 2 | 3\*2 + (−1) = 5 | 3\*5 + 3 = 18 | 3\*18 + 1 =55 | 3\*55 + (−5) =160 |

Bảng 5.1

Vậy p(3) = 160.

Khi so sánh các mục nhập của bảng với công thức (6.3), ta thấy rằng:

3. 2 + (−1) = 5 là giá trị của 2x - 1 tại x = 3

3. 5 + 3 = 18 là giá trị của x (2x - 1) + 3 tại x = 3

3. 18 + 1 = 55 là giá trị của x (x (2x - 1) + 3) + 1 tại x = 3,

và cuối cùng, 3. 55 + (−5) = 160 là giá trị của x (x (x (2x - 1) + 3) + 1) - 5 = p (x) tại x = 3.

5.5.2 Mô tả thuật toán

|  |
| --- |
| //Đánh giá một đa thức tại một điểm nhất định theo quy tắc của Horner  //Input: Một mảng P [0...n] gồm các hệ số của đa thức bậc n, được lưu trữ từ thấp nhất đến cao nhất và một số x  //Output: Giá trị đa thức tại x  P ← P[n]  **for** i ← n − 1 **downto** 0 **do**  p ← x ∗ p + P[i]  **return** p  Số phép nhân và số phép cộng được cho bởi cùng một tổng:  M(n) = A(n) = |

5.5.3 Triển khai bằng python 3

|  |
| --- |
| **def** **horner**(poly, n, x):  result = poly[0]  **for** i **in** range(1, n):     result = result\*x + poly[i]  **return** result  poly = [2, -6, 2, -1]  x = 3  n = len(poly)  **print**("Value of polynomial is " , horner(poly, n, x)) |

5.6 Problem Reduction (Giảm thiểu vấn đề)

5.6.1 Ý tưởng thuật toán

Một thuật toán hiệu quả hơn nhiều để tính toán bội số ít phổ biến nhất có thể được nghĩ ra bằng cách sử dụng giảm thiểu vấn đề. Có một thuật toán (thuật toán Euclid) để tìm ước số chung lớn nhất, là một tích của tất cả các thừa số nguyên tố chung của m và n. Chúng ta có thể tìm thấy một công thức liên quan lcm (m, n) và gcd (m, n)? Không khó để thấy rằng tích của lcm (m, n) và gcd (m, n) bao gồm mọi thừa số của m và n đúng một lần và do đó đơn giản là bằng nhau thành tích của m và n. Quan sát này dẫn đến công thức:

lcm(m, n) = m . n / gcd(m, n)

5.6.2 Mô tả thuật toán

|  |
| --- |
| **def** **lcm**(x,y):  **if** x > y:  result = x  **else**:  result = y  **while**: |

5.6.3 Triển khai bằng python 3

|  |
| --- |
| #Tìm ước chung lớn nhất  **def** **gcd**(a,b):  **if** a == 0:  **return** b  **return** gcd(b % a, a)   # Tính bội chung nhỏ nhất  **def** lcm(a,b):  **return** (a / gcd(a,b))\* b |

**TÀI LIỆU THAM KHẢO**

Introduction to the Design and Analysis of Algorithms (Levitin)