Notes on CS-3329:

Analysis of Algorithms

Instructor: GS. TS. Suman Kumar

Course Section:	**CS-3329-IHFA**
Title:	Analysis of Algorithms
Credits:	3
Final Grade:	A

Troy University Resources

Big – O (upper bound)

Một hàm $f(n)$ được gọi là $O(g(n))$ nếu tồn tại hai hằng số dương $c > 0$, $n_0 > 0$ sao cho với mọi $n \geq n_0$:

$$0 \leq f(n) \leq c\, g(n)$$

Big – Omega (lower bound)

Định nghĩa toán học: Hàm $f(n)$ được gọi là $\Omega(g(n))$ nếu tồn tại các hằng số dương c và n_0 sao cho với mọi $n \geq n_0$:

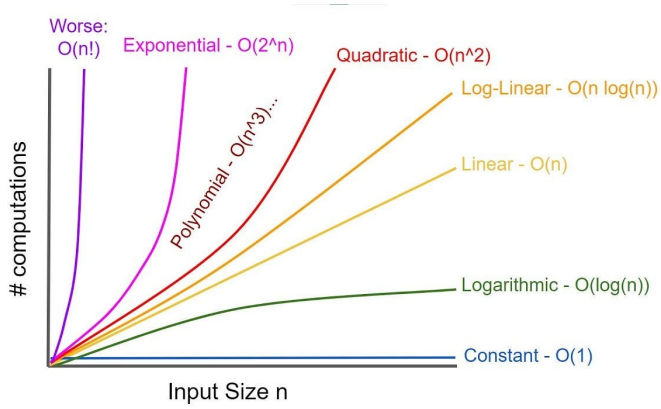
$$0 \leq c\, g(n) \leq f(n)$$

Big – Theta (tight bound)

Định nghĩa toán học: Hàm $f(n)$ được gọi là $\Theta(g(n))$ nếu tồn tại các hằng số dương c_1, c_2, n_0 sao cho với mọi $n \geq n_0$:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

- **Commonly used notation:** $f(x) = O(g(x))$
- **Correct notation:** $f(x) \in O(g(x))$



Insertion Sort

Dùng các ký hiệu trên, ta bắt đầu phân tích thời gian chạy của Insertion Sort.

Pseudocode:

```
Insertion-Sort(A)
for j = 1 to n-1
    i = j

    while (i > 0 and A[i] < A[i-1])
        swap(A[i], A[i-1])
        i = i - 1
```

Để chuyển từ mã giả sang công thức thời gian chạy, ta gán chi phí cho các thao tác.

t_a = time assignment

t_c = time comparison

`for j = 1 to n-1`	$t_a(n-1)$
`i = j`	$t_a(n-1)$
`while (i > 0 and A[i] < A[i-1])`	$\leq 2t_c(k+n-1)$
`swap(A[i], A[i-1])`	$3kt_a$
`i = i - 1`	kt_a

Gọi k là tổng số lần thân `while` chạy. Nhưng nhớ là mỗi vòng `for`, điều kiện `while` vẫn phải check thêm một lần để biết lúc nào dừng. Vậy tổng số lần check là $k+n-1$, mỗi lần tốn nhiều nhất 2 phép so sánh.

Công thức tổng quát

Ta vẫn lấy t_a là thời gian cho một phép gán, t_c là thời gian cho một phép so sánh. Cộng cả mấy lần check `while` bị thiếu lúc nãy thì ta có cận trên:

$$T(n) \leq (2t_a + 2t_c)(n-1) + (4t_a + 2t_c)k$$

Tức là $T(n) = O(n+k)$. Mà `for` vẫn chạy $n-1$ lượt, còn `swap` phải làm k lần, nên cũng có $T(n) = \Omega(n+k)$. Gộp lại:

$$T(n) = \Theta(n+k)$$

Giờ cứ thay k theo từng case là ra.

Troy University Resources

Best case (đã tăng sẵn)	While không chạy nên $k = 0$ (không swap lần nào), nhưng điều kiện `while` vẫn phải check $n - 1$ lần. $$T_{\text{best}}(n) = \Theta(n)$$ $k = 0$ (không có phép hoán đổi)
Worst case (mảng giảm dần hoàn toàn)	Nên: $$\begin{aligned} k &= 1 + 2 + 3 + \cdots + (n-1) \\ &= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} = \frac{n^2}{2} - \frac{n}{2}. \end{aligned}$$ $k = \Theta(n^2)$ $T_{\text{worst}}(n) = \Theta(n^2)$
Average case	$\Theta(n^2)$

Divide and conquer -> giới thiệu vào Merge Sort

Best case	$\Theta(n \log n)$
Average case	$\Theta(n \log n)$
Worst case	$\Theta(n \log n)$

Thuật toán này không ổn vì

Space = $O(n)$

Merge Sort làm hai việc chính:

1. Chia mảng thành hai nửa: $2T(n/2)$.
2. Trộn hai mảng đã sắp xếp: tối đa $n - 1$ phép so sánh, xấp xỉ n.

Vì vậy:

$$T(n) = 2T(n/2) + n - 1$$

Cận trên thường dùng:

$$T(n) < 2T(n/2) + n.$$

Ta chia mảng liên tục đến khi $\frac{n}{2^k} = 1$, nghĩa là k = log(n)

Sau k lần mở truy hồi:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$\begin{aligned} T(n) &= 2^{\log n} T(1) + n \log n \\ &= n \cdot 0 + n \log n \\ &= n \log n. \end{aligned}$$

Pseudocode:

```
merge-sort(A, beg, end)
    if beg < end
        mid = floor((beg + end) / 2)
        merge-sort(A, beg, mid)
        merge-sort(A, mid + 1, end)
        merge(A, beg, mid, end)
```

Hàm merge dưới đây là bước ghép hai phần đã sắp xếp trong thuật toán trên.

```
merge(A, beg, mid, end)
    sizeL = mid - beg + 1
    sizeR = end - mid

    L = copy of A[beg .. mid]
    R = copy of A[mid+1 .. end]

    i = 1
    j = 1
    k = beg

    while i <= sizeL and j <= sizeR
        if L[i] <= R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        k = k + 1
   while i <= sizeL
        A[k] = L[i]
        i = i + 1
        k = k + 1

    while j <= sizeR
        A[k] = R[j]
        j = j + 1
        k = k + 1
```

Ký hiệu	**Ý nghĩa**
i	Con trỏ đang xét trong mảng L
j	Con trỏ đang xét trong mảng R
k	Con trỏ ghi kết quả vào mảng A

Troy University Resources

Quick Sort

Quick Sort cũng dùng chia để trị; phần dưới xét các trường hợp pivot chia mảng.

Worst Case (số favourite chọn làm lệch L và R) như kiểu mảng đã sort	$T(n) = T(n-1) + n$ $T(n) = O(n^2)$
Special Case, tức là pivot không chia đều nhưng cũng không quá lệch.	$T(0.1n) + T(0.9n) + n$ $T(n) = O(n \log n)$
Best case (chia đều 2 bên trục)	$T(n) = 2T(n/2) + n$ $T(n) = \Omega(n \log n)$
Average case	$T(n) = O(n \log n)$

Algorithm	**Extra Array**	**Stack**	**Total Space**
Merge Sort	$O(n)$	$O(\log n)$	$O(n)$
Quick Sort	$O(1)$	$O(\log n)$ (trung bình)	$O(\log n)$

```
quickSort(A, beg, end)
if beg < end
    p = partition(A, beg, end)
    quickSort(A, beg, p - 1)
    quickSort(A, p, end)


partition(A, beg, end)

pivot = A[end]

i = beg
j = end

while true
    while A[i] < pivot
        i = i + 1
    while A[j] > pivot
        j = j - 1

    if i < j
        swap(A[i], A[j])
    else
        return i
```

Heap | priority queue

Ta chuyển sang heap, cấu trúc sẽ được dùng để xây dựng Heap-sort ở phần sau.

Đầu tiên nó là:

1, Left-full Binary Tree

Nó có các cấu hình trong số 1 như:

2,

- min-heap, Mỗi parent $\leq$ child
- Max-heap, Mỗi parent $\geq$ child
- Insertion -> số phép so sánh = h $\approx$ log(n)
- Deletion, xóa root, ta lấy value dưới cùng copy lên root, heapify -> số phép so sánh = h $\approx$ log(n)

parent $= (i-1)/2$

left $= 2i+1$

right $= 2i+2$

Đây là công thức để xây lại cây từ 1 mảng

Cách build a heap given n values:

1. Repeated insertion (cách này chỉ nói vào thôi chứ không ai build cách này)

$T(n) = \log(1) + \log(2) + \cdots + \log(n)$

$T(n) = \log(n!)$

$= O(n \log n)$

2. **Bottom-Up Build Heap (dùng cách này)**

$\Rightarrow$ put the items in the array

$\Rightarrow$ Run heapify from bottom to top

$T(n) = O(n)$

Tại sao T(n)=O(n):

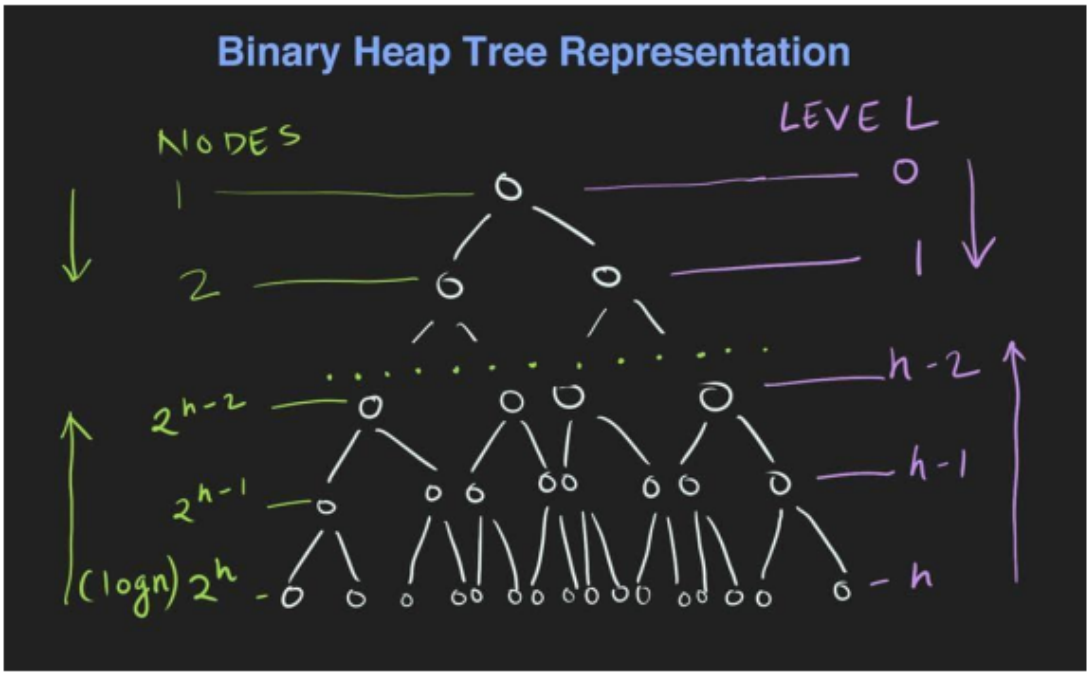
Cây nhị phân có n items thì tầng dưới cùng có n+1/2 items là lá

Number of compare in buildings heap:

$$\frac{n+1}{2} \cdot 0 + \frac{n+1}{4} \cdot 1 + \frac{n+1}{8} \cdot 2 + \cdots + \frac{n+1}{2^{h+1}} \cdot h$$

Triển khai dần thì nó sẽ cho chúng ta O(n)

Troy University Resources



Heap-sort

Max-heap (Min-heap cũng thế)

Lặp lại cho đến khi heap còn 1 phần tử

1. **Đổi chỗ root với phần tử cuối cùng** trong heap
   → phần tử lớn nhất được đưa về đúng vị trí cuối mảng.
2. **Giảm kích thước heap đi 1**
   → vì phần tử cuối đó đã được sorted rồi, không tính vào heap nữa.
3. **Heapify tại root**
   → để khôi phục lại tính chất max-heap cho phần còn lại.

Mỗi lần hết logn nhân với n – 1 lần nên ra O(nlogn)

Total time to sort using a heap = time to build heap + time for recursive heapify = O(n) + O(nlogn) = O(nlogn)

Lower bound on comparison based sorting

Sau các thuật toán trên, ta xét giới hạn chung của việc sắp xếp bằng so sánh.

Số trường hợp tối thiểu có thể xảy ra >= n!, số lá tối đa <= 2 mũ h (chiều cao cây)

Counting sort

Giới hạn vừa nêu dành cho cách sắp xếp bằng so sánh; Counting Sort chuyển sang đếm các giá trị.

Pseudocode

```
CountingSort(A)
    n = size(A)
    k = max(A)

    create array B[0..k]
    create array C[0..n-1]
    for i = 0 to k
        B[i] = 0

    for i = 0 to n-1
        B[A[i]] = B[A[i]] + 1

    for i = 1 to k
        B[i] = B[i] + B[i-1]

    for i = n-1 downto 0
        C[B[A[i]] - 1] = A[i]
        B[A[i]] = B[A[i]] - 1
    return C
```

$\Rightarrow$ Chạy k+1 lần => O(k)

$\Rightarrow$ Chạy n lần => O(n)

$\Rightarrow$ Chạy k lần => O(k)

$\Rightarrow$ Chạy n lần => O(n)

$$\begin{aligned} T(n) &= O(k) + O(n) + O(k) + O(n) \\ &= O(n+k). \end{aligned}$$

Counting Sort rất nhanh khi:

$$k = O(n)$$

khi đó:

$$T(n) = O(n)$$

$\rightarrow$ **linear time**

Nếu:

n = 100

k = 1,000,000

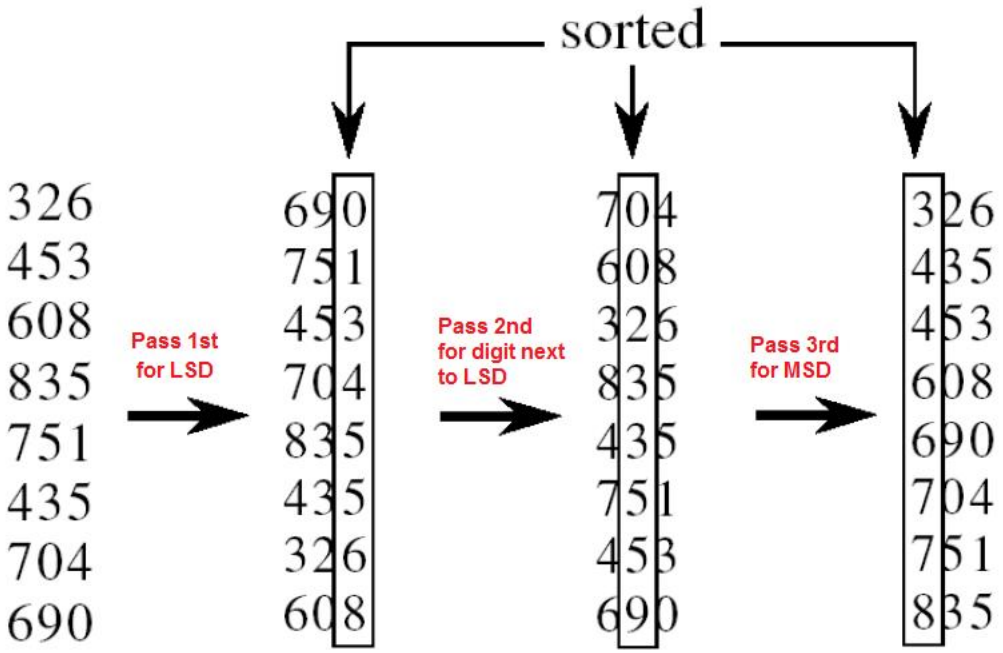
thì:

$$T(n) = O(n+k) \approx O(k)$$

$\rightarrow$ rất tốn bộ nhớ.

Troy University Resources

radix sort



Bản chất của radix sort là sort theo từng cột digit, ví dụ:

Bước 1	**Bước 2**	**Bước 3**
$170 \rightarrow 0$	0 : 170, 90	170, 90, 802, 2, 24, 45, 75, 66
$45 \rightarrow 5$	2 : 802, 2	
$75 \rightarrow 5$	4 : 24	
$90 \rightarrow 0$	5 : 45, 75	
$802 \rightarrow 2$	6 : 66	
$24 \rightarrow 4$		
$2 \rightarrow 2$		
$66 \rightarrow 6$		

Không so sánh chỉ là lấy từng cột ra nhóm các số vào

Radix sort dùng Counting Sort cho bước 2 -> O(n+k)

Tổng thời gian

Vì thuật toán chạy d lần và mỗi lần tốn $O(n+k)$, nên:

$$\begin{aligned} T(n) &= d \times O(n+k) \\ &= O\big(d(n+k)\big). \end{aligned}$$

Chạy d lần cho d cột (chục, đơn vị, trăm…)

Và vì k chỉ chạy từ 0 -> 9 nên T(n) = O(d(n + 10)) ≈ O(dn), gần như linear time -> very good

Median-of-Medians

Đến đây, mục tiêu đổi từ sắp xếp cả mảng sang tìm một phần tử theo hạng.

Ta có mảng A gồm n phần tử.

Ta muốn tìm phần tử có hạng r; còn tìm median thì là tìm cái ở hạng $\lceil n/2 \rceil$ (hạng tính từ 1).

```
FindRank(A, r)       // n = |A|, 1 <= r <= n
if n <= 5: sort(A) and return A[r-1]
1. Chia n phần tử thành các nhóm, mỗi nhóm tối đa 5 phần tử
2. Find median của từng group bằng việc sorting
3. B = array chứa các median của từng nhóm
4. x = FindRank(B, ceil(|B|/2)) // median-of-medians
5. AL, AE, AR = partition3(A, x) // < x, = x, > x
6. if r <= |AL|:
       return FindRank(AL, r)
   else if r <= |AL| + |AE|:
       return x
   else:
       return FindRank(AR, r - |AL| - |AE|)
```

Sort một group tối đa 5 phần tử, worst case tốn $O(5^2) = O(1)$ thôi. Nên sort hết đống group (khoảng $n/5$ nhóm) là $O(n)$:

$\Rightarrow 5^{2*}\frac{n}{5} = 5\text{n}$

$\Rightarrow \text{T(n/5)}$

Ta xem case xấu nhất là phải rẽ tiếp vào bên có nhiều phần tử nhất. Nhờ chọn pivot bằng median-of-medians, với n đủ lớn, có ít nhất $n/4$ phần tử $\leq x$ và ít nhất $n/4$ phần tử $\geq x$ (chứng minh bằng cách đếm các group 5 phần tử). Vậy khi đã tách riêng những phần tử $= x$, bên phải đệ quy tiếp có tối đa:

$$\max(|A_L|, |A_R|) \leq \frac{3n}{4}$$

So tất cả tốn: tìm median trong B (cỡ $n/5$ phần tử), rồi có thể phải đệ quy tiếp bên có tối đa $3n/4$ phần tử. Mấy việc còn lại như chia nhóm, sort từng group, partition đều tuyến tính. Bỏ qua phần làm tròn, gom chi phí đó thành cn thì:

$$T(n) \leq T(n/5) + T(3n/4) + cn$$

Nhớ là không được gộp $T(n/5) + T(3n/4)$ thành $T(19n/20)$ nhé. Nó là hai nhánh đệ quy khác nhau, nên phải tính tổng việc của cả hai nhánh.

Sau k lần

Giả sử ta xét đến mức k của cây đệ quy. Tổng size của các bài toán con ở mỗi mức giảm theo tỷ lệ $\frac{1}{5} + \frac{3}{4} = \frac{19}{20}$ (bỏ qua phần làm tròn khi xét tiệm cận). Vậy công việc ngoài đệ quy ở mức k tối đa là:

$$W_k \leq cn \left(\frac{19}{20}\right)^k$$

Troy University Resources

Đến đây thì cộng hết các mức lại. Nó là cấp số nhân có công bội $19/20 < 1$, nên:

$$\begin{aligned} T(n) &\le cn \sum_{k=0}^{\infty} \left(\frac{19}{20}\right)^k + O(n) \\ &= 20cn + O(n) = O(n). \end{aligned}$$

Vậy ta đã prove được worst case nó ra $O(n)$ -> tốt.

Master Theorem

Là phương pháp tìm nhanh cận giới hạn của phương trình đệ quy:

$$T(n) = aT(n/b) + f(n)$$

Trong đó:

- n: kích thước của bài toán đầu vào.
- $a \ge 1$: số lượng bài toán con được chia ra ở mỗi bước đệ quy.
- $b > 1$: hệ số thu nhỏ kích thước của mỗi bài toán con, có kích thước n/b.
- $f(n)$: chi phí thời gian (asymptotically positive) cho công việc bên ngoài các lời gọi đệ quy, bao gồm chia bài toán và kết hợp kết quả.

3 trường hợp:

Case 1: Leaves dominate

Nếu $f(n) = O\big(n^{\log_b a - \varepsilon}\big)$ với $\varepsilon > 0$ thì:

$$T(n) = \Theta\big(n^{\log_b a}\big)$$

Case 2: Balanced

Nếu $f(n) = \Theta\big(n^{\log_b a} \log^k n\big)$ với $k \ge 0$ thì:

$$T(n) = \Theta\big(n^{\log_b a} \log^{k+1} n\big).$$

Ý nghĩa: Chi phí mỗi mức trong cây đệ quy tương tự nhau.

Case 3: Root dominates

Nếu $f(n) = \Omega\big(n^{\log_b a + \varepsilon}\big)$ và thỏa điều kiện chính quy

$$af(n/b) \le cf(n), \qquad c < 1,$$

thì:

$$T(n) = \Theta(f(n))$$

Quy trình giải nhanh

Khi gặp $T(n) = aT(n/b) + f(n)$:

1. **Bước 1:** Tính $n^{\log_b a}$.

2. **Bước 2:** So sánh với $f(n)$ và xác định trường hợp thích hợp:

So sánh	Case	Kết quả
$f(n)$ nhỏ hơn	1	$\Theta(n^{\log_b a})$
Tương đương	2	$\Theta(n^{\log_b a} \log n)$
$f(n)$ lớn hơn	3	$\Theta(f(n))$

Pattern Matching

Tiếp theo là bài toán trên chuỗi: tìm vị trí xuất hiện của một mẫu trong văn bản.

Cách 1: vét cạn

Best case: n-m+1 = lower bound(n) (luôn fail ở kí tự đầu)

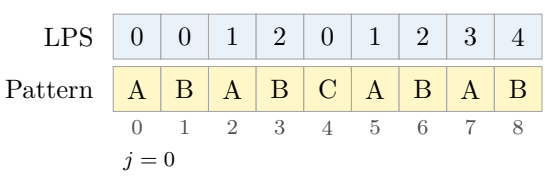
Worst case: đến kí tự cuối mới fail (n-m+1) nhân m = O(n.m)

Cách 2: Knuth–Morris–Pratt (KMP)

Từ cách vét cạn, ta xét cách tận dụng phần đã khớp để tiếp tục tìm kiếm.

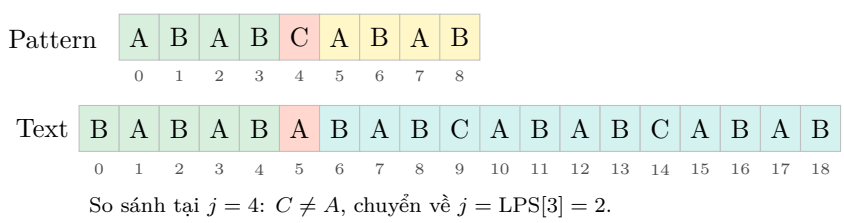
"Longest proper prefix = suffix of the matched portion"

Câu nói trên nghĩa là phần tiền tố mà đã tìm được, vô hình chung lại thành đuôi mới mà ta bắt tìm từ đuôi đấy.



Giả sử đến C không match thì ra fall back về trước nó là index 3 đã match, ta phải về lại index 2 vì AB đã lặp dài đến hết AB một lần khỏi phải check làm gì nữa.

Ta phải quay lại cái giống đầu nhất vì có khả năng đây có thể là đầu mới 😊, éo biết được. Đấy là cái hay của thuật toán



Ví dụ đến C đỏ thì phải check lại từ j = 2.

Nói đến thời gian:

1. Xây bảng tham chiếu longest prefix = suffix hết O(m), với m là độ dài pattern vì phải duyệt qua hết pattern và 1 con trỏ theo dõi prefix = suffix, tổng là ≤ 2m thì là O(m)

Troy University Resources

2. Trong lúc search:

- i là con trỏ trên text
- j là con trỏ trên pattern

i tăng nhiều nhất n (số lượng kí tự trong text)

j luôn tăng nhỏ hơn n vì làm éo gì có chuyện tìm pattern to hơn cả văn bản gốc

$\Rightarrow \leq 2n$ thì là O(n)

Pseudocode:

Gồm 2 bước là build cái bảng longest prefix = suffix hay còn gọi là bảng pi và tìm kiếm

```
ComputePi(P)
m = length(P)
π[0] = 0
j = 0

for i = 1 → m-1
    while j > 0 and P[i] ≠ P[j]
        j = π[j-1]

    if P[i] == P[j]
        j = j + 1

    π[i] = j

return π


KMP(T, P)
n = length(T)
m = length(P)

π = ComputePi(P)
j = 0

for i = 0 → n-1
    while j > 0 and T[i] ≠ P[j]
        j = π[j-1]

    if T[i] == P[j]
        j = j + 1

    if j == m
        print(i - m + 1)     // found match
        j = π[j-1]
```

Graph algorithms

Sau bài toán trên chuỗi, ta chuyển sang mô hình đồ thị để biểu diễn các quan hệ.

Graph = **đồ thị = tập đỉnh (V) + cạnh (E)**

Ví dụ 1 vấn đề có thể giải quyết bằng graph:

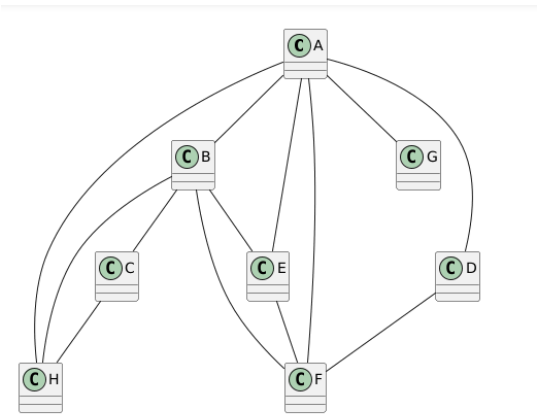
Bạn có **các module đào tạo**:

A, B, C, D, E, F, G, H

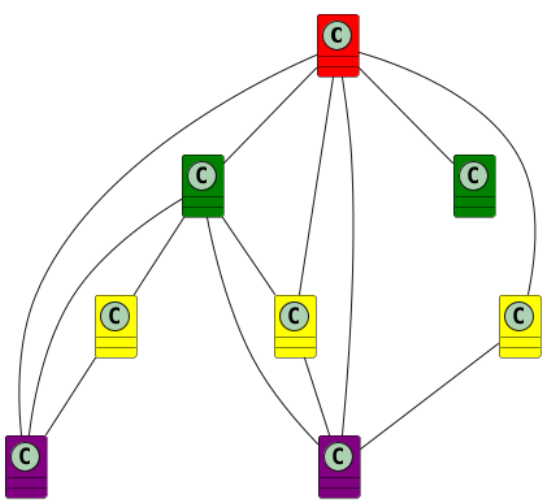
Một số **cặp module không thể có cùng nhân viên**, nghĩa là **các cặp này xung đột**:

A-B, B-C, A-D, A-E, B-E, E-F, A-F, B-F, D-F, A-G, A-H, B-H, C-H

Bạn muốn **xếp lịch tối thiểu** sao cho các module xung đột không diễn ra cùng một slot (buổi đào tạo).



Có nối nhau thì nghĩa là cùng 1 buổi có xung đột, cách làm bài này chính là ta sẽ tô màu vào từng đỉnh sao cho 2 đỉnh có nối thì không cùng màu là không xung đột, sau đấy các đỉnh cùng màu dồn vào cùng 1 ngày vì có xung đột éo đâu



Tô xong ra 4 màu 4 buổi end game, quá dễ

Troy University Resources

Navigation on graph

Sau ví dụ tô màu, ta xét cách duyệt qua các đỉnh của đồ thị.

Duyệt đồ thị có 2 cách: Breadth-First Search (BFS) và Depth-First Search (DFS)

BFS: dùng khi "gần → xa", cần đường đi ngắn nhất. (unweighted graph)

Pseudocode:

```
BFS(V, E, s):
    for each u in V - {s}:
        d[u] = ∞
        parent[u] = NIL
    d[s] = 0
    parent[s] = NIL
    Q = empty queue
    ENQUEUE(Q, s)

    while Q not empty:
        u = DEQUEUE(Q)
        for each v in Adj[u]:
            if d[v] = ∞:
                d[v] = d[u] + 1
                parent[v] = u
                ENQUEUE(Q, v)
```

Mỗi đỉnh được **enqueue và dequeue tối đa một lần**.

$O(|V| \cdot 1 + |V| \cdot 1) = O(|V|)$

Vì mỗi đỉnh chỉ được dequeue **một lần**, mỗi cạnh cũng chỉ được xét **một lần** (hoặc hai lần với vô hướng).

$O(|E|)$

$\Rightarrow O(|V|) + O(|E|) = O(|V| + |E|)$

DFS: đi sâu nhất xuống 1 nhánh rồi quay lui lên trên đi nhánh khác

$\Rightarrow$ Stack:

```
DFS-Stack(G, s):
    create empty stack S
    push s into S
    visited[u] = false for all u ∈ V

    while S is not empty:
        u = pop(S)

        if visited[u] == false:
            visited[u] = true
            time = time + 1
            d[u] = time

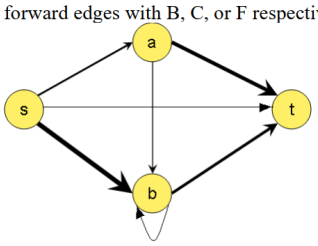
            for each v ∈ Adj[u]:
                if visited[v] == false:
                    classify edge (u,v) as Tree Edge
                    push v into S
                else:
                    if visited[v] == true (GREY logic):
                        classify edge (u,v) as Back/Forward/Cross Edge based on d/f

            time = time + 1
            f[u] = time
```

$O(|V|) + O(|E|) = O(|V| + |E|)$

Nếu đề bài ra chỉ là show DFS thì game quá easy, nhưng nếu ra kiểu:

7. Run the DFS on the following graph, with s being the source. Identify the back, cross or forward edges with B, C, or F respectively. 8 points.



Sẽ khó hơn vì phải xây DFS spanning tree.

Ví dụ xây cây này ở câu trên thì ta đi từ

s -> a (tree), a->t (tree) không đi được nữa về lại a

a->b (tree), b -> b (back), b->t (forward)

sau đó thấy t không đi đâu được, back về lại a

a đã đi hết 2 hướng về lại s, s -> b không đi vì b đã khám phá rồi (forward)

s sang t cũng không đi vì t đã khám phá rồi (forward)

Troy University Resources

Bản chất ta sẽ vẽ được cái spanning xong ta sẽ chỉ ra được 4 loại cạnh:

Tree Edge: v được phát hiện lần đầu từ u

Back Edge: v là tổ tiên của u (self-loop luôn là back edge), nếu quay lại cha thì không tính là back

Forward Edge: v là con cháu của u (nhưng không phải tree edge)

Cross Edge: u và v không có quan hệ tổ tiên – con cháu

Minimum spanning tree (MST)

Từ việc duyệt đồ thị, ta chuyển sang bài toán tìm cây khung có tổng trọng số nhỏ nhất.

spanning tree là cây khung, có trọng số giữa các đường, liên thông, vô hướng

1. Prim (Hợp vào graph dày):

```
Prim(G, r)
    for each u ∈ V[G]
        key[u] = ∞
        parent[u] = NIL

    key[r] = 0
    Q = V[G]

    while Q ≠ ∅
        u = Extract-Min(Q)

        for each v ∈ Adj[u]
            if v ∈ Q and w(u, v) < key[v]
                parent[v] = u
                key[v] = w(u, v)
                Decrease-Key(Q, v, key[v])

    return parent
```

Này khởi tạo mảng lên cho đống này hết O(|V|)

Build heap từ |V| phần tử: O(|V|)

$\Rightarrow$ Q = Min-Heap chứa tất cả các đỉnh (theo key)

Extract này mỗi lần mất O(|V|log |V|)

Decrease-Key: O(|E| log |V|)

$O(|V|) + O(|V|) + O(|V| \log |V|) + O(|E| \log |V|)$

$= O((|V| + |E|) \log |V|)$

2. Kruskal (hợp với graph thưa):

```
Kruskal(G)
    for each vertex v ∈ V[G]
        Make-Set(v)

    sort edges E by increasing weight

    MST = ∅

    for each edge (u, v) ∈ E
        if Find(u) ≠ Find(v)
            MST = MST ∪ {(u, v)}
            Union(u, v)

    return MST
```

$\Rightarrow O(|V|)$

$\Rightarrow$ Sắp xếp cứ cho là chuẩn O(nlogn) thì đây sắp đỉnh thì O(|E|log|E|)

Các thao tác Union và Find set rất nhanh nên $\approx$ O(|E|)

Thêm vào MST -> O(|V|) (vì MST có |V|-1 cạnh)

$\Rightarrow O(|V|) + O(|E| \log |E|) + O(|E|) + O(|V|)$

$= O(|E| \log |E|)$

Single source shortest path

Vẫn trên đồ thị có trọng số, ta chuyển sang tìm đường đi ngắn nhất từ một đỉnh xuất phát.

1. Bellman-Ford: áp dụng cho cả cạnh âm

```
BellmanFord(G, w, s)
    for each vertex v ∈ V[G]
        d[v] = ∞
        parent[v] = NIL

    d[s] = 0

    for i = 1 to |V[G]| - 1
        for each edge (u, v) ∈ E[G]
            if d[v] > d[u] + w(u, v)
                d[v] = d[u] + w(u, v)
                parent[v] = u

    for each edge (u, v) ∈ E[G]
        if d[v] > d[u] + w(u, v)
            return FALSE

    return TRUE
```

Time complexity:

$$\boxed{T(|\,V\,|, |\,E\,|) = O(|\,V\,| \cdot |\,E\,|)}$$

Troy University Resources

2. Dijkstra: chỉ áp dụng cho mỗi cạnh dương

```
Dijkstra(G, w, s)
    for each vertex v ∈ V[G]
        d[v] = ∞
        parent[v] = NIL

    d[s] = 0

    Q = V[G]

    while Q ≠ ∅
        u = Extract-Min(Q)

        for each v ∈ Adj[u]
            if d[v] > d[u] + w(u, v)
                d[v] = d[u] + w(u, v)
                parent[v] = u
                Decrease-Key(Q, v, d[v])

    return d, parent
```

$\Rightarrow$ priority queue theo d[v]

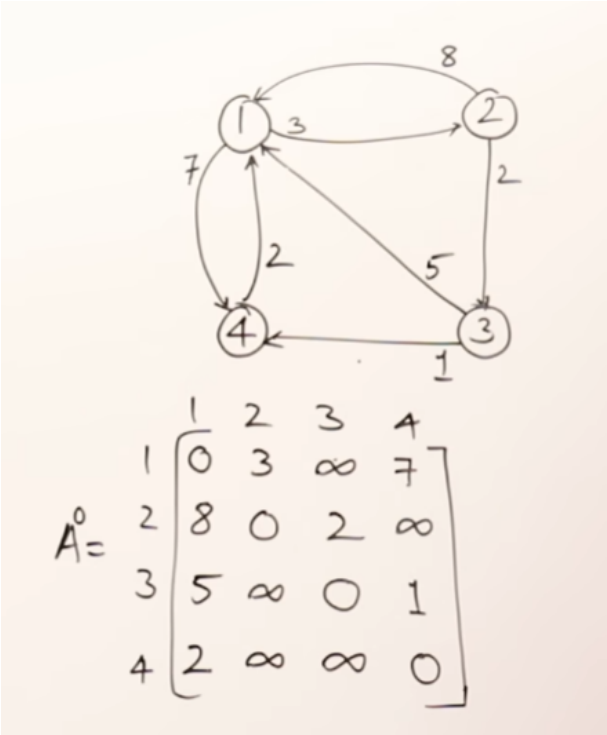
Time complexity:

$O((|V| + |E|) * \log |V|)$

All-Pairs Shortest Path

Từ một đỉnh xuất phát, ta mở rộng yêu cầu sang tìm đường đi ngắn nhất giữa mọi cặp đỉnh.

1) Floyd–Warshall: dense (best suited)



```
FloydWarshall(W)
    n = number of vertices

    D = W

    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                D[i][j] = min(D[i][j], D[i][k] + D[k][j])

return D
```

- Bản chất thằng này là sẽ tìm đường đi từ đỉnh nào đến đỉnh khác, tìm full các khả năng xong thành cái ma trận
- Cái số mũ trên đầu của cái tên ma trận kia biểu thị đi qua đỉnh vào, ví dụ A^0 nghĩa là không đi qua đỉnh trung gian nào
- Giả sử A^2 thì phải ví dụ ta cần tìm từ 1->4 thì phải xem giá trị 1->4 hiện tại với đi cái đường 1->2+2->4 xem có ngắn hơn không
- Làm lần lượt từ mũ 0 đến mũ max ở đấy ví dụ ở đây là 4 thì A^0, A^1, A^2, A^3, A^4

Pseudocode có 3 vòng lặp lồng nhau:

Mỗi vòng chạy n lần

$\Rightarrow \boxed{T(n) = O(n^3)}$

2) Johnson: sparse

```
Johnson(G):
    add vertex q
    for each v in V:
        add edge (q, v) weight = 0

    h, negativeCycle = BellmanFordDistances(G, q)
    if negativeCycle:
        return FALSE

    for each edge (u, v):
        w(u,v) = w(u,v) + h(u) - h(v)

    for each vertex u in V:
        d_prime = Dijkstra(G, u)
        for each vertex v in V:
            d[u][v] = d_prime[v] - h(u) + h(v)

    return d
```


Troy University Resources

- Đầu tiên ta thêm 1 đỉnh giả q,
- sau đó ta tính h từ q đến các đỉnh, tức $h(v)$ là khoảng cách ngắn nhất từ q đến v

$$h(q) = 0, \qquad h(v) = \delta(q, v) \quad (v \in V).$$

Mấy cái $h(v)$ này phải chạy Bellman–Ford mới ra, không phải cứ cho tất cả bằng 0 nhé. Có cạnh trọng số âm thì $h(v)$ cũng có thể âm.

- Chạy Bellman–Ford 1 lần từ q để lấy các giá trị $h(v)$ (không phải $|V|$ lần)
- Sau đó sẽ có các cái h đã tính, dùng chúng để đổi trọng số
- $w'(u, v) = w(u, v) + h(u) - h(v)$ (đổi trọng số)
- Sau đó chạy Dijkstra từ từng đỉnh
- Sau đó đổi về khoảng cách thật (theo trọng số ban đầu):

$d(u, v) = d'(u, v) - h(u) + h(v)$

Time complexity (Dijkstra dùng *binary heap*):

$$\boxed{O\big(|V||E| + |V|(|V| + |E|)\log|V|\big)}$$

Dynamic Programming

Sau các thuật toán trên đồ thị, ta xét thêm cách xây dựng lời giải từ các bài toán con qua những ví dụ quy hoạch động.

- **Tính Fibonacci:**

Dynamic

```
F(n):
    if n <= 2:
        return 1

    dp[1] = 1
    dp[2] = 1

    for i = 3 to n:
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Đệ quy

```
F(n):
    if n <= 2:
        return 1

    return F(n-1) + F(n-2)
```

- **Knapsack problem:**

```
KNAPSACK(weight, value, n, W)
    Create array dp[0..n][0..W]

    for w = 0 to W
        dp[0][w] = 0

    for i = 1 to n
        for w = 0 to W
            if weight[i] > w
                dp[i][w] = dp[i-1][w]
            else
                dp[i][w] = max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]])

    return dp[n][W]
```

- **Matrix chain multiplication**

```
MATRIX-CHAIN-ORDER(p, n)
    Create array m[1..n][1..n]

    for i = 1 to n
        m[i][i] = 0

    for L = 2 to n
        for i = 1 to n - L + 1
            j = i + L - 1
            m[i][j] = ∞

            for k = i to j - 1
                q = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]

                if q < m[i][j]
                    m[i][j] = q

    return m[1][n]
```

- **rod-cutting**

cho 1 thanh độ dài n, ở mỗi độ dài thì giá bán khác nhau, ví dụ:

n = 4

p[1] = 1

p[2] = 5

p[3] = 8

p[4] = 9

Bản chuẩn

Troy University Resources

```
ROD-CUTTING(price, n)
    let dp[0..n] be a new array
    dp[0] = 0

    for j = 1 to n
        best = -∞
        for i = 1 to j
            best = max(best, price[i] + dp[j - i])
        dp[j] = best

    return dp[n]
```

Biến thể có thêm cost mỗi lần cut

```
MODIFIED-ROD-CUTTING(price, n, c)
    let dp[0..n] be a new array
    dp[0] = 0

    for j = 1 to n
        best = price[j]
        for i = 1 to j - 1
            best = max(best, price[i] + dp[j - i] - c)
        dp[j] = best

    return dp[n]
```

Complexity theory

Các phần trước xét cách giải từng bài toán; giờ ta chuyển sang phân loại độ khó của bài toán.

P (Polynomial Time): bài toán giải được trong **polynomial time**. **Giải được nhanh**

NP (Nondeterministic Polynomial Time): Các bài toán mà nếu có đáp án rồi thì kiểm tra đáp án nhanh được.

- Khi nói NP-complete -> output chỉ là **YES/NO**
- Còn phiên bản tối ưu thường hay được xếp là NP-hard.

Reduction:

Nếu có hai bài toán A và B, viết:

$$A \leq_p B$$

nghĩa là muốn chứng minh B "khó", ta lấy một bài đã biết khó là A, rồi reduce A về B. cái viết kiểu trên chỉ xảy ra khi đã chứng minh được B khó ít nhất là bằng A

NP-hard là tập các bài toán khó ít nhất ngang với mọi bài toán trong NP.

Nghĩa là: với mọi bài toán A trong NP, ta có thể reduce A về B(với B là bài toán NP-hard). Do đó, nếu giải được B thì có thể suy ra lời giải của A.

NP-complete:

1. nếu nó thuộc NP
2. là NP-hard

Quy trình chuẩn để chứng minh một bài toán mới là NP-complete

Bước 1: Chứng minh bài toán thuộc NP

Tức là:

- nếu câu trả lời là YES
- và kiểm tra lời giải đó trong polynomial time

Bước 2: Chọn một bài đã biết NP-complete

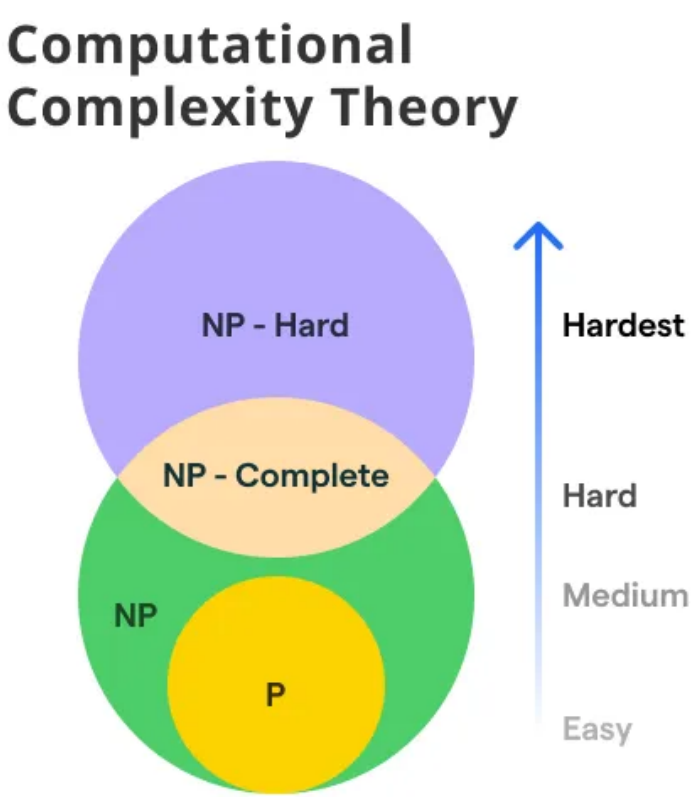
Bước 3: Reduce bài đã biết khó sang bài mới

Phải là:

$$\text{Known NP-complete problem} \leq_p \text{New problem}$$

Bước 4: Kết luận

- vì bài cũ reduce sang bài mới → bài mới là NP-hard
- lại có bài mới thuộc NP
- nên bài mới là NP-complete



Troy University Resources

Approximation Algorithm

Sau khi xét độ khó của bài toán, ta chuyển sang hướng tìm lời giải gần đúng.

Khi gặp mấy bài như:

Traveling Salesman Problem

Subset Sum Problem

→ không có thuật toán nhanh (polynomial)

→ brute force = thử tất cả → $O(2^n)$ hoặc tệ hơn

Không cần lời giải chính xác

👉 Chỉ cần gần đúng

Approximation Ratio

- OPT (Optimal solution, lời giải tốt nhất) = 100
- thuật toán cho ra = 120

→ vẫn OK 👍

$$\text{ratio} = \frac{\text{solution của thuật toán}}{\text{optimal solution}}$$

Thuật toán 2-approximation cho Traveling Salesman Problem (TSP)

Bước:

1. Tạo MST (Minimum Spanning Tree)
2. DFS trên cây đó
3. Ghi lại thứ tự đi
4. Bỏ các đỉnh lặp (shortcut) chỉ giữ lần xuất hiện đầu tiên của mỗi đỉnh.
5. nối đỉnh cuối quay về đỉnh xuất phát để tạo chu trình

Kết quả: chu trình thu được có tổng chi phí không vượt quá 2 lần chi phí tối ưu.

Prove cách này là 2-approximation

1. Gọi T là MST ⇒

$$w(T) \leq OPT$$

2. Đi DFS trên T⇒ mỗi cạnh đi 2 lần ⇒

$$cost = 2w(T)$$

3. Shortcut (bỏ node trùng) dùng triangle inequality ⇒

cost không tăng, nghĩa là đi bình thường thì hết 2w(T) nếu đi shortcut thì "có thể" sẽ ngắn hơn nên sẽ thành ≤

⇒ $cost(ALG) \leq 2w(T)$

⇒ $cost(ALG) \leq 2OPT$

subset sum

Có thể tạo ra tổng = t bằng cách cộng vài số trong S không?

$S = \{1, 3, 4, 5\}, \quad t = 8$

Bài toán này có thể giải chính xác bằng dynamic programming trong thời gian $O(nt)$ khi các số trong S và mục tiêu t đều là số nguyên không âm. Đây là độ phức tạp giả đa thức (pseudo-polynomial), vì nó phụ thuộc vào giá trị của t, không chỉ số bit biểu diễn t. Khi t lớn, người ta có thể dùng approximation algorithm nếu chấp nhận sai số.

Thay vì giữ quá nhiều tổng gần giống nhau, ta chỉ giữ một vài tổng đại diện.

nếu hai số quá gần nhau

thì giữ một số thôi

bỏ số kia đi

Nhờ vậy list không to.

```
APPROX-SUBSET-SUM(S, t, ε)
    n = |S|
    L0 = ⟨0⟩

    for i = 1 to n do
        Li = MERGE-LISTS(Li−1, Li−1 + xi)
        Li = TRIM(Li, ε / (2n))
        remove from Li every element that is greater than t

    return the largest element in Ln


TRIM(L, δ)
    let L' be a new list
    append first element of L to L'

    last = first element of L

    for each element y in L starting from the second element do
        if y > last × (1 + δ) then
            append y to L'
            last = y

    return L'
```

Giả sử vừa quyết định **giữ** một số z.

Thì mọi số y thỏa:

$$y \leq (1 + \delta)z$$

đều bị xem là **quá gần** z, nên không cần giữ nữa.

Chỉ giữ lại z làm **đại diện**.

Dạng này không hỏi pseudocode mà sẽ kiểu

$S = \langle 2, 5, 6, 9 \rangle$

$t = 15$

Troy University Resources

$\varepsilon = 1$

APPROX-SUBSET-SUM(S, t, ε)

kết luận output cuối cùng

Cách làm:

Ban đầu phải xác định được $\delta = \varepsilon$ / (2n)

$L_0 = \langle 0 \rangle$

x1=2, Cộng 2 vào từng phần tử của L0

merge với L0 = <0,2>

ta coi z là số trước của y ta cần check xem (1+ δ) nhân z có lớn hơn y không, nếu có thì bỏ

cứ như thế xong chuyển sang bước tiếp theo

Set cover

Cuối cùng là bài toán chọn tập để bao phủ phần tử và cách chứng minh NP-complete.

chọn cái nào cover được nhiều thứ CHƯA làm nhất

U = {A, B, C, D, E} (chưa làm gì hết)

C = {} (chưa chọn gì)

Một số k

$S_1 = \{A, B, C\}$

$S_2 = \{B, D\}$

$S_3 = \{C, D, E\}$

$S_4 = \{E\}$

Chọn $\leq$ **k tập con** sao cho:

(tức là cover hết mọi phần tử trong U)

Bài toán này quá dễ nên khả năng sẽ thành 1 bài để tính NP-complete chẳng hạn

5. [3+4] Explain NP completeness. Define set cover problem and explain how it is NP complete.

Đầu tiên prove được là nó thuộc NP, quá dễ cho 1 lời giải bất kì có thể check ngay được

Ta chọn ngay vertex cover là bài toán NP-complete xong map nó

Ta thấy chọn 1 vertex cũng như chọn 1 tập

Các cạnh cover cũng như các đỉnh cover trong 1 tập

$\Rightarrow$ Set cover cũng chính là NP-hard