(12) · BST (Portion-1)

```
INORDER-Recursive (x)
  if x != NIL
    INOR-Recursive ( x.left)
    print x.key.
    Inorder-recc (x.right).
```

```
Preorder(x)
if x != NIL
print x.key
  pre (x.left)
  pre ( x.right).
```

```
INORDER-NONRECURSIVE (T)
  S = [ ]
  done = 0
  current = T.root
  while ! done
    if current != NIL
        PUSH( S, current)
        current = cur.left.
    else if !is empty ()
        current = pop(S)
        print current
        current = current.right.
    else
        done = 1
```

BST → left subtree < right subtree

Min heap → that but parent in minimum the both left & Right subtree.

min heap not guarantee to print the key in sorted order in linear time because we have no way to know which subtree contains the next min. elect-

Portion-1

$I = I_1 + I_2$ → No. of internal, at least one child.

① $I_2 = L - 1$

Inducti. node: **Base:** $L = 1$ ⇒ $I_2 = 0$

i.e. $I = I_1$ at least one child.

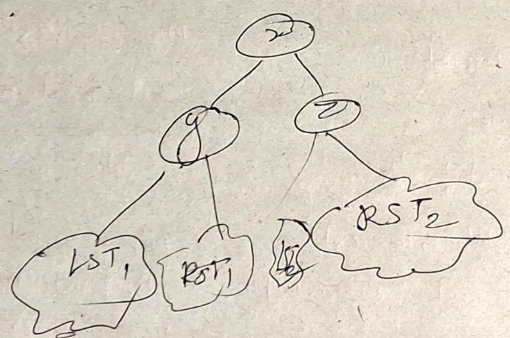
Inductive Step: Binary tree Total nodes = N+1

Add leaf node — Dual 2nd child

$I_2 = L - 1$ ⇒ $I_2 = L - 1$

12-2-9

②



LST : Root ≥ LST

RST : Root ≤ LST.

③ INORDER-TREE-WALK (x)
 if x ≠ NIL
 INORDER-TREE-WALK (x.left)
 (print x.key) (x.right)

RBST → Insert $O(\lg n)$

Total insert = $n \times (\lg n)$
 $= n \lg n$

Inorder tree = $O(n)$

* Best case : $O(n \lg n)$.

Worst case : → Insert = $O(n)$ as linked-list

Total insert = $n \times (O(n)) = O(n^2)$ — sorted

Inorder = $O(n)$

6 chapter

④ Max-heap in linear time (bottom-up approach) Root Max >> left & Right child.

· Max-heap property →

* We can build a Max-heap from unordered array in linear time $O(n)$ using a process called heapify.

→ The key insight is that we can start from the bottom of tree (the last non-leaf node & apply the heapify operation to each node.

Parent = $\lfloor \frac{i}{2} \rfloor$, left child = $2i$ & Right child = $2i+1$

last non-leaf node is at index = $\lfloor \frac{n}{2} \rfloor - 1$,

MAX-HEAPIFY (A, n, i):
 largest = i
 left = 2i+1 // left child index
 right = 2i+2 // right child index
 if left < n && A[left] > A[largest]
  largest = left
 if right < n && A[right] > A[largest]
  largest = right
 if largest ≠ i
  A[i] ⟺ A[largest] ; MAX-HEAPIFY (A, n, largest)

$T(n) = \sum_{i=0}^{n/2 - 1} O(\lg n)$ which ⇒ $T(n) = O(n)$

BUILD-MAX-HEAP (A)
 n = length. A
 for i to $\lfloor \frac{n}{2} \rfloor - 1$, -1, -1
  MAX-HEAPIFY (A, n, i)

Start MAX-HEAP(A) from the last non-leaf node $\lfloor \frac{n}{2} \rfloor - 1$

⑤ To merge k sorted lists $O(n \log k)$, n → total no. of elements in all the k lists.

↳ min-heap
↳ priority queue.

Algorithm
↳ ① Create min-heap
↳ ② Insert first element → $O(\log k)$
↳ ③ Extract min & Insert next element: → $O(\log k)$ } $O(n \log k)$
↳ ④ Repeat until all elements from the lists have been taken.

```
Merge_k_lists (lists):
    min_heap = [ ]
    result = [ ]
    for i in range(length of lists):          → O(n)
        if lists[i]:
            heapq.heappush(min_heap, (lists[i][0], i, 0))   → log(k)

    while min_heap:                                  → log(k).
        val, list_index, elem_index = heapq.heappop(min_heap)
        result.append(val)
        if elem_index + 1 < len(lists[list_index]):
            next_val = lists[list_index][elem_index+1]
            heapq.heappush(min_heap, (next_val, list_index, elem_index+1))   → log(k)
    return result
```

Total = $O(n \log k)$

⑥ To check whether 2 array have the same set of no. in $O(n)$.

Approach
↳ ① count the frequency of each no.: Use hash map (dictionary.)
↳ ② Compare the Frequency map.

Steps:
① Create 2 hash map.
② Populate the hash map
↳ A_1 → increment +1 in 1st hash map.
↳ A_2 → increment +1 in 2nd hash map.
③ Compare the hash maps: freq_map1 == freq_map2
i.e. they have same key with same counter. (Should be identical)

Time complexity
* Count frequencies → $O(n)$ → length of each array
* Inserting element ⇒ $\theta(1)$.
* Comparing 2 hash map takes $O(k)$ → No. of unique elements in the array (which ⇒ $O(n)$).

$\therefore O(n) + \theta(1) + O(n)$

⇒ $T(n) = O(n)$

12.1-3

(7) Nonrecursive — Inorder (T)

```
S = [ ]
current = T.root
done = [ ]
While ! done
    if current ≠ NIL
        push (S, current)
        current = current.left
    else if ! S.EMPTY( )
        pop(S) = current
        print "current"
        current = current.right
    else
        done = 1
```

Portion - 2 Chapter 3 of 4

3.2-5

(2) $\lg(\lg^* n)$ or $\lg^*(\lg n)$, which one greater.

$\lg^* n$ → iterative logarithm funct^n

$$\lg^* n = \begin{cases} 0 & ; \text{ if } n \le 1 \\ 1 + \lg^*(\lg n) & ; \text{ if } n > 1 \end{cases}$$

i.e. The growth of $\log^* n$ is extremely slow.

Ex:- $\lg^*(2) = 1$

$\log^* 4 = 1 + \lg^*(\lg 4)$

$= 1 + \lg^*(2)$

$= 1 + 1 = 2$

We have, $\lg^* 2^n = 1 + \lg^* n$

$$\lim_{n\to\infty} \frac{\lg(\lg^* n)}{\lg^*(\lg n)} = \lim_{n\to\infty} \frac{\lg(\lg^* 2^n)}{\lg^*(\lg 2^n)}$$

$$= \lim_{n\to\infty} \frac{\lg(1 + \lg^* n)}{\lg^* n}$$

$$= \lim_{n\to\infty} \frac{\lg(1+n)}{n}$$

$$= \lim_{n\to\infty} \frac{1}{1+n}$$

$$\to 0$$

Therefore, we have that $\lg^*(\lg n)$ is asymptotically larger.

So!

(a) $2^{\lg n}$, $(\lg n)^{\lg n}$, e^n, $4^{\lg n}$, $(n+1)!$, $\sqrt{\lg n}$

$2^{\lg n} \to n^{\lg 2} \to n$

$(\lg n)^{\lg n} \to 2^{(\lg n)(\lg\lg n)}$ → Faster than polynomial & slower than exponential e^n

$4^{\lg n} \to n^{\lg 4} \to n^2$

$(n+1)!$ → faster than exponential. using stirling's approx. $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$\sqrt{\lg n} = (\lg n)^{\frac{1}{2}}$ → very slow

Hence, $(n+1)! > e^n > n^2 > 2^{\lg n} > (\lg n)^{\lg n} > \sqrt{\lg n}$

(b) $\left(\frac{3}{2}\right)^n$, n^3, $\lg^2 n$, $\lg(n!)$, $\left(2^{2^n}\right)$, $n^{\frac{1}{\lg n}}$

$\left(\frac{3}{2}\right)^n$ → exponential funct^n faster but slower than 2^{2^n}

n^3 → polynomial funct^n of degree 3.

$\lg^2 n$ → slower than polynomial

$\lg(n!)$ → Stirling approximat^n by $(n!) \approx n\lg n - n$ faster than n^3 & slower than exponential

$\left(2^{2^n}\right)$ → double exponential

$n^{\frac{1}{\lg n}}$ → slower than poly & logarithmic & faster than

$n\lg n > n^3$

$2^{\log^* n} \times 2^{\log^* n} = 2^{2\log^* n}$

$2^{2\log^* n} < 2^{\log^* n}$

Hence, $2^{2^n} > \left(\frac{3}{2}\right)^n > \lg(n!) > n^3 > n^{\frac{1}{\lg n}} > \lg^2 n$

Answer (5)

(a) $T(n) = 2T(\sqrt{n}) + \lg n$

Let's perform a change of variable to simplify the recurrence

Let $n = 2^m \Rightarrow \lg n = m$ & $\sqrt{n} = 2^{m/2}$

$\therefore T(2^m) = 2T(2^{m/2}) + m$

Let $S(m) = T(2^m)$

$\Rightarrow \boxed{S(m) = 2S(m/2) + m}$ using master theorem,

$a = 2,\ b = 2,\ k = 1,\ p = 0$

$a = b^k$, $2 = 2$ } using case 2 $T(m) = \theta(m^{\log_b a} \log^{p+1} m)$

$= \theta(m^{\log_2 2} \log^{0+1} m)$

$\boxed{S(m) = \theta(m \log m)}$

Now, $T(2^m) = S(m) = \theta(m \lg m)$ & $m = \log n$

then $T(n) = \theta(\log n \cdot \lg \log(n))$

i.e. $T(n) = \theta(\log n \cdot \lg(\log(n)))$ Ans.

(b) $T(n) = T(n-2) + \frac{1}{\lg n}$

Solve using iterative approach

$T(n) = T(n-2) + \frac{1}{\lg n}$

$T(n-2) = T(n-4) + \frac{1}{\lg(n-2)}$

$T(n-4) = T(n-6) + \frac{1}{\lg(n-4)}$

$\vdots \qquad \vdots$

$T(n) = T(n-2k) + \sum_{k=0}^{k-1} \frac{1}{\lg(n-2i)}$

stopping condition; when $n-2k$ is a sufficiently small constant say (1) Thus $k \approx n/2$

Approximating the terms,

$\sum_{i=0}^{k-1} \frac{1}{\lg(n-2i)} \leq \sum_{i=0}^{n/2} \frac{1}{\lg(n/2)} = \frac{n}{2} \cdot \frac{1}{\lg n}$

Thus, $T(n) = \theta\left(\frac{n}{\lg n}\right)$ Ans.

(c) $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

Using a change of variables, Let $n = 2^m \Rightarrow \log n = m$ & $\sqrt{n} = 2^{m/2}$

$\therefore T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m$

Let $S(m) = T(2^m)$ then,

$S(m) = 2^{m/2} S(m/2) + 2^m$

Divide by 2^m $\quad \frac{S(m)}{2^m} = \frac{S(m/2)}{2^{m/2}} + 1$

Let $U(m) = \frac{S(m)}{2^m}$, so we have $U(m) = U(m/2) + 1$

Using master theorem $U(n) = U(n/2) + 1$

$a = 1, b = 2, k = 0$ & $p = 0$

$a = b^k$, $1 = 1$ } Case 2: $U(n) = \theta\left(n^{\log_b a} \lg^{p+1} n\right)$

$$\boxed{U(n) = \theta(\lg n)}$$

Thus, $S(n) = 2^{n/2} S(n/2) + 2^n \Rightarrow \boxed{S(n) = U(2^n)}$

$\therefore S(n) = \theta(n \cdot 2^n)$ since $\boxed{n = \lg n}$

$\therefore T(2^n) = 2^{n/2} T(2^n/2) + 2^n$ } $S(n) = T(2^n)$

$T(n) = \theta(n \lg n)$

$\boxed{T(n) = \theta(n \lg \lg n)}$ Ans.

(d) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

→ At the top level 0, the cost is n

→ At level 1, we have three subproblems of sizes $n/2$, $n/4$, $n/8$.

Thus, the total cost is: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} = n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right) \leq \frac{7n}{8}$

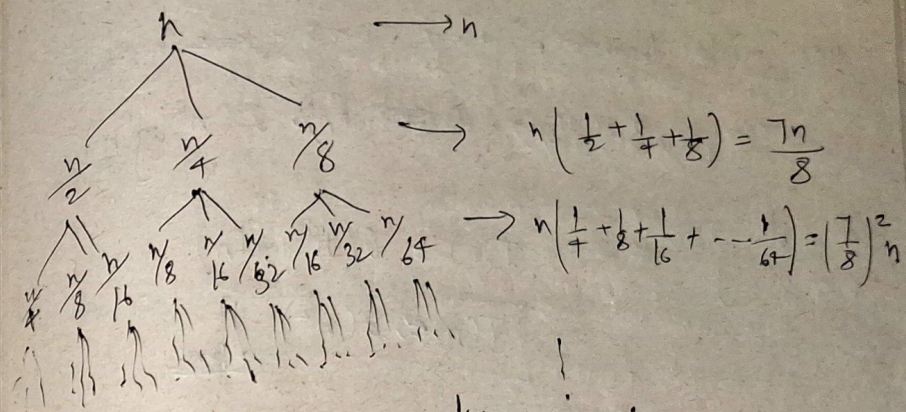
→ This pattern continues, where the total cost at each level is $n \cdot \left(\frac{7}{8}\right)^k$

→ The recurrence has a logarithmic height of $\log_2 n$, since $n \to n/2 \to n/4 \to \cdots$

∴ Total cost is a geometric series

$$T(n) = n \sum_{k=0}^{\lg n} \left(\frac{7}{8}\right)^k = \theta(n)$$

OR



$$\sum_{i=1}^{\lg n} \left(\frac{7}{8}\right)^i n = \theta(n)$$ Ans.

OR

We use substitution method, $T(n) \leq cn$ for some constant $c > 0$

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$

$\leq cn/2 + cn/4 + cn/8 + n$

$\leq \frac{7}{8} cn + n$

$\leq (1 + \frac{7}{8}c)n$

$\leq cn$ $(\because c \geq 8)$

$\therefore \boxed{T(n) = O(n)}$

Similar for lower bound $\boxed{T(n) = \Omega(n)}$

Hence, $\boxed{T(n) = \theta(n)}$ Ans.

(c) $T(n) = 7T(n/3) + n^2$

Using master theorem, $a = 7$, $b = 3$ & $k = 2$ & $p = 0$

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$

$a < b^k$, $7 < 3^2$ } Case 3: $T(n) = \Theta(n^k \log^p n)$

$\boxed{T(n) \approx \Theta(n^2)}$ Ans

(d) Show that $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$

* First part: The sum of the n term over all levels

$$\sum_{k=0}^{\lg n} n = n \lg n$$

Second part: The sum of $17k * 2^k$ terms

$$\sum_{k=0}^{\lg n} 17k * 2^k = O(n)$$

$\therefore T(n) = O(n \lg n) + O(n) \Rightarrow \boxed{O(n \lg n) = T(n)}$ Proved

OR

$T(n) \approx 2T(n/2) + n$

$a = 2, b = 2$, $a = b^k$ } Case 2: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

$= \Theta(n^{\log_2 2} \log^1 n)$

$\boxed{T(n) = \Theta(n \lg n)}$

OR

A level k:

→ There are 2^k calls to $T\left(\frac{n}{2^k} + 17k\right)$

→ Cost for each call is $\frac{1}{2}\left(\frac{n}{2^k} + 17k\right)$

→ Total cost at each level $= 2^k \cdot \frac{1}{2}\left(\frac{n}{2^k} + 17k\right) = \frac{n}{2} - \frac{17}{2} 2^k$

$\therefore T(n) = \sum_{k=0}^{\lg n} \left(\frac{n}{2} + \frac{17}{2} 2^k\right)$

$= \sum_{k=0}^{\lg n} \frac{n}{2} + \frac{17}{2} \sum_{k=0}^{\lg n} 2^k$

$\leq \frac{n}{2} \lg(n)$

$\therefore T(n) \leq O(n \lg(n))$ Ans.

(5) (a)

```
RANDOM-SEARCH (X, A, n)
    V = ϕ
    while |ϕ| ≠ n
        i = RANDOM (1, n)
        if A[i] = X
            return i
        else
            V = V ∪ i
    return NIL
```

N can be implemented in multiple ways: A hash table, a tree, or a bitmap.

(b) RANDOM-SEARCH is well-modelled by Bernoulli trails. The expected number of picks is n.

(c) In similar fashion, the expected number of picks is n/k.

(d) This is modelled by balls & bins problem, $n(\ln n + O(1))$.

(e) The worst case running time is n. The avg. $\frac{n+1}{2}$

(f) The worst-case running time is $n-k+1$.

The avg. $\frac{(n+1)}{(k+1)}$.

Let X_i be an indicator random variable that ith element is a match. $Pr\{X_i\} = \frac{1}{k+1}$.

Let Y be an indicator random variable that ~~can be~~ we have found a match after the first $n-k+1$ elements. $(Pr\{Y\} = 1)$.

Thus, $E[X] = E[X_1 + X_2 + \cdots X_{n-k+1}]$

$$= 1 + \sum_{i=1}^{n-k} E[X_i]$$

$$= 1 + \frac{n-k}{k+1}$$

$$= \frac{n+1}{k+1}$$

Ans.

(g) both the worst case & avg-case is $O(n)$

(h) It's the same as DETERMINISTIC-SEARCH, only replace avg-case with expected.

(i) Definitely DETERMINISTIC-SEARCH, ~~only we replace "avg-case"~~. SCRAMBLE-SEARCH give better expected results. but for the cost of the randomly permuting the array, which is a linear operation. In the same time we have scanned the full array & reported a result.

* Expected No. of Tosses:

(b) For the n^{th} ball tossed, the probability that it lands in new bin (i.e. ~~One that has not yet has a ball~~ one that has not yet got hit)

$$\frac{b-(n-1)}{b}$$

⇒ If landing in bin that already has a ball

$$1 - \frac{b-(n-1)}{b} = \frac{n-1}{b}$$

Ans

Harmonic relation.

i.e. $E[X] = \sum_{n=1}^{b} \frac{b}{b-(n-1)} = b \sum_{n=1}^{b} \frac{1}{n}$

$\approx b \cdot \approx \ln(b) + \gamma + O(\frac{1}{b})$

$$E[X] = bH_b \approx b(\ln b + \gamma)$$

$$\boxed{E[X] = b \ln b}$$

(7) Expected No. of Empty bins

$$Y_i = \begin{cases} 1, & \text{if th ith bin is empty} \\ 0, & \text{otherwise} \end{cases}$$

→ The probability of specific ball does not go into bin $i = \frac{n-1}{n}$.

→ Since the tosses are independent, the prob. that none of the balls go to into bin i is

$$Pr(X_i = 1) = \left(\frac{n-1}{n}\right)^n$$

Then

$$E[X_i] = Pr(X_i = 1) = \left(\frac{n-1}{n}\right)^n$$

Total No. of empty bins

$$X = \sum_{i=1}^{n} X_i$$

By the linearity of expetation

$$E[X] = \sum_{i=1}^{n} E[X_i] = n \cdot \left(\frac{n-1}{n}\right)^n$$

$$\left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e} \quad \text{where, } e \approx 2.718$$

$$\therefore E[X] \approx n \cdot \frac{1}{e} = \frac{n}{e}$$

$$\Rightarrow \boxed{E[X] \approx \frac{n}{e}}$$

(2) Expected No. of bins with exactly one ball

$$Y_i = \begin{cases} 1, & \text{if the ith bin has exactly one} \\ 0 & \text{otherwise} \end{cases}$$

$$Pr(Y_i = 1) = \frac{1}{n} \cdot \left(\frac{n-1}{n}\right)^{n-1}$$

$$E[Y_i] = Pr(Y_i = 1) = \frac{1}{n}\left(\frac{n-1}{n}\right)^{n-1}$$

$$Y = \sum_{i=1}^{n} Y_i$$

By linearity of expectation

$$E[Y] = \sum_{i=1}^{n} E[Y_i] = n \cdot \frac{1}{n} \cdot \left(\frac{n-1}{n}\right)^{n-1}$$

$$E[Y] = \left(\frac{n-1}{n}\right)^{n-1}$$

Approximation $\left(1 - \frac{1}{n}\right) \approx \frac{1}{e}$

$$E[Y] \approx \frac{1}{e} \Rightarrow \boxed{E[Y] \approx 1}$$

⑧ Longest streak $L(n)$ is max. value of k for which there is at least one streak of length k.

We want to find the value $k \Rightarrow \frac{n}{2^k} \geq 1$

$2^k \approx n$

Taking log both side

$$k = \log_2(n)$$

$$E[L(n)] \approx \log_2(n)$$ Ans.

Portion → 3

8.1-4

① → merging sorted sequences typically requires $\Omega(n \lg m)$ comparisons, where m is the no. of subsequences.

In this case, $m = \frac{n}{k} \Rightarrow$ lower bound merging $\Omega(n \lg \frac{n}{k})$

→ This simplifies to $\Omega(n \lg k)$, as merging n/k subsequences required at least $\Omega(n \lg k)$ comparisons.

Or

8.1-4 Assume that we need to construct a binary decision tree to represent comparison -

k subsequence → $(k!)^{n/k}$ possible permutations.

height of DT is 2^h i.e.

$$(k!)^{n/k} \leq 2^h$$

Taking log both sides

$$h \geq \frac{n}{k} * \lg(k!) \geq \frac{n}{k} * \left(\frac{k \ln k - n}{\ln 2}\right)$$

$$h = \Omega(n \lg k)$$ Ans.

8.3-3

② * Loop Invariant : AT the begining of the for loop, the array is sorted on the last $i-1$ digits.

* Initialization: The array is trivially sorted on the last 0 digits.

* Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the ith digit, the last i digits, sorted the array.

→ It is obvious that elements with diff. digit in the ith position are ordered o/c; in the case of the same ith digit, we still get a correct order, because we are using a stable sort & the elements were already sorted on the last $i-1$ digits.

* Termination: The loop terminates when $i = d+1$. Since the invariant holds, we have the numbers sorted on d digits.

8.3-4

(a) → First run through the list of integers & convert each one to base n, then radix sort them.

→ Each number will have at most $\log_n n^3 = 3$ digits.

→ So there will only need to be 3 passes.

→ For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

16.1-4

(4) Greedy Algorithm for Activity Scheduling / Interval-graph coloring problem

Sorting → ① Sort activities by start time

min-heap → ② Initialize the lecture halls based on the priority queue (min-heap).

Greedy choice → ③ Assign activities to lecture halls.

Hence, The size of the heap at the end will give the min. number of lecture halls required.

Total time = $O(n \lg n)$

16.2-3

(5) 0-1-knapsack (n, W)

Initialize an $(n+1)$ by $(W+1)$ table K

for i=1 to n

$K[i,0] = 0$

for j = 1 to W

$K[0,j] = 0$

for i = 1 to n

for j = 1 to W

if $j < i.weight$

$K[i,j] = K[i-1,j]$

else

$K[i,j] = \max\{K[i-1,j], K[i-1, j-i.wt] + i.value\}$.

16.2-5

(6) Consider the leftmost interval. It will do no good if it extends any further left than the leftmost point. However, we know that it must contain the leftmost point.

So, we know that it's contain left hand side is exactly the leftmost point.

So, we just remove any point that is within a unit distance of the leftmost point. Since they are contained in this single interval.

Then, we just repeat until all points are covered.

Since at each step there is a clearly optimal choice for where to put the leftmost interval, this final sol'n is optimal.

16.4-4

Ans (a) Alternate sol^n:

→ Maintain a set of free (but already used) lecture halls F & currently busy lecture halls B.

→ Sort the classes by start time. For each new start time which you encounter, remove a lecture hall from F, schedule the class in that room & add the lecture hall to B.

→ If F is empty, add new unused lecture hall to F.

→ When a class finishes, remove its lecture hall from B & add it to F. This is optimal for following reason,

→ Suppose we have just started using the mth lecture hall for the first time. This only happens when every classroom ever used before is in B.

→ But this means that there are m classes occuring simultaneously, so it is necessary to have m distinct lecture halls in use.

16.3-8

(7) → For any 2 characters, the sum of their frequencies exceeds the frequency of any other character, so initially Huffman coding makes 128 small trees with 2 leaves each.

→ At the next stage, no internal node has a label which is more than twice that of any other,

⇒ so we are in the same setup as before. Continuing in this fashion, Huffman coding builds a complete binary tree of height $\lg 256 = 8$, which is no more efficient than ordinary 8-bit length codes.

OR

8-bit fixed length code ⇒ 256 characters.

$\log 256 = 8 \Rightarrow 2^8 = 256$

Sol^n.

(8) To prove that the closest-point heuristic for TSP returns a tour whose cost is at most 2 the cost of optimal tour. Use $d(a,c) \leq d(a,b) + d(b,c)$

→ Triangle inequality

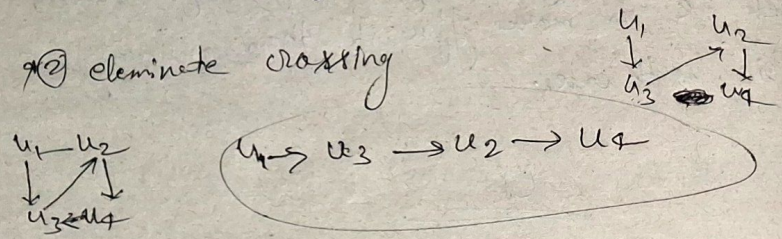
→ The properties of the heuristic

Based on Triangle inequality heuristic provides a 2-approximation for TSP

$$C_h \leq 2C^*$$

(9) ① Assume the optimal tour crosses itself.
i.e. (u_1, u_2) & (u_3, u_4) → 2 edges crosses itself.

② eliminate crossing



③ Using triangle inequality for Euclidean distances guarantees that the new path cannot have a longer total length than the original.

$$c(u_1, u_3) + c(u_2, u_4) \leq c(u_1, u_2) + c(u_3, u_4)$$

This is because, in Euclidean geometry, the direct path b/w 2 points is always shorter or equal to any detour path & swapping edges avoids crossing.

④ Contradict^n: If the original tour had a crossing, we found a way to rearrange the tour without increasing the cost. This contradiction the assumption that the original tour was optimal. Therefore, the optimal tour cannot have any crossing.

(10) Single-source shortest path → Bellman-Ford
→ Dijkstra's Algo.

& V → Vertices, E → edges, (u,v) → each edge & w(u,v) → weights.

① Graph representation as a Matrix (W) & distance vector d:

$W = |V| \times |V|$, & $w_{u,v} = \begin{cases} w(u,v) & \to \text{Direct connt of 2 vertices} \\ \infty & \to \text{No connectn.} \end{cases}$

$d_s = |V|$

$d_s = \begin{cases} 0 & ; \text{Initially} \\ \infty & ; \text{to all other vertices } i \end{cases}$

② Relaxation as matrix–vector product

↳ Update rule

$$d_i^{(k+1)} = \min_{j \in V} \left(d_j^{(k)} + w_{ji} \right)$$

apply on matrix-vectors, we get

$$D^{(0)} = d^{(0)}$$

$$d^{(k+1)} = \min \left(d^{(k)} + W \right)$$

③ Evaluation of Bellman-Ford like Algo.

→ Step(1): Initialization of distance vector $d^{(0)}$, where $d_s^{(0)} = 0$ & all other entries are ∞.

→ Step(2): Relaxation $d^{(k+1)} = \min \left(d^{(k)} + W \right)$

→ Step(3): Repeat $|V|-1$ times.

Time complexity = $O(|V|^3)$.

↳ $W = |V| \times |V|$ } $O(|V|^3)$
Repeat $= |V| + 1$ } total times.

Portion - 1

① $G = T$ proved

→ conversely, suppose I/p graph is an undirected graph & connected but not tree. Then G must contain a cycle c.

→ Suppose c consists of k nodes $v_1, v_2, \dots v_k$ i.e. c is the cycle $v_1 \to v_2 \to v_3 \dots v_k \to v_1$.

→ Now, in DFS tree nodes $v_1, v_2 \dots v_k$ will all be same path from root to a leaf. Suppose v_f is 1st of these nodes to be visited. Then remaining nodes must be visited at some point during explore(v_f). ∴ v_i are all strongly connected. However BFS tree nodes will form at least 2 branches, branching from 1st visited node.

Hence, BFS & DFS will be same iff the I/p graph is a tree.

② DFS

→ ① Run DFS on graph, let DFS as tree (T).

→ ② If $G = T$ (all of edges of G are in T), return no

→ ③ Otherwise, Let (u,v) be an arbitrary edge of G that is not in T.

→ ④ Find path from u to v in T, let p denotes path.

→ ⑤ Return $p \cup \{(u,v)\}$.

DFS

① Iterate overall the nodes of the graph G & keep visited array to track the visited nodes.

② Run DFS
- Set visited[current] as 1
- Iterate over all adjacent nodes of the current node in the adjacency list
  - If it is not visited then Run DFS on that node & return true if it returns true
  - Else if the adjacent node is visited & not the parent of current node then return true.
- Return False.

23.1-1&3

(8) Proof based on the cut property of MST.

Using cut (partition of the vertices into two disjoint subsets) in the graph, the minimum-weight edge crossing that cut must be part of some MST.

Proof: → Assume (u,v) is minimum-weight edge in the graph G. & w(u,v) is weight of edg (u,v).

→ Consider a cut that separates the vertex u in one subset S & vertex v in the other subset V/S.

→ This cut divides the vertices of G into 2 disjoints subs

$S = \{u\} \cup A$ & $V/S = \{v\} \cup B$, where A&B are subsets of vertices.

OR

Suppose that A is an empty set of edges. Then, make any cuts that has (u,v) crossing it. Then, since that edge is of minimal weight, we have that (u,v) is a light edge of that cut, & so it is safe to add.

Since, we added it, then once we finish we constructed the trees, we have that (u,v) is contained in a MST.

(9) 23.1-8

We want to show that all MSTs of graph have the same set of edge weights, even if the actual edge may differ. The proof based on the (cut property & cycle property of MST)

G → (V, E) → undirected graph

T → MST of G

T' → any other MST of G

L → sorted list of edge weights of T

→ Total No of edges exactly n−1

→ Sum of weights $w(T) = w(T')$

→ L_T → sorted list of edge weights of T

→ $L_{T'}$ → sorted list of edge weight of T'

→ Both T & T' have n−1 edges.

∴ When sorted L_T & $L_{T'}$ must be identical.

Hence, All MST of a graph that same total weight, & since each MST has the same no. of edges (n−1), their edge weights must appear in the same multiset.

Therefore, their sorted lists of edges weights must be identical.

(5) Kruskal's

Book

```
MST-KRUSKAL(G, w)
    A = φ
    for each vertex v ∈ G.V        } O(V)
        MAKE-SET(v)
    Sort the edges of G.E into
    non-decreasing order by
    weight w.
    for each edge (u,v) ∈ G.E      } O(E)
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u,v)}
            UNION(u, v)
    return A.
```

Total $T(n) = O((V+E)\alpha(V))$

$|E| \geq |V|-1$ then

Disjoint-set operation is $O(E\alpha(V))$

Since $\alpha(V) = O(\lg V) = O(\lg E)$

$\therefore T(n) = O(E \lg E)$

$\because |E| < |V|^2$ then

$T(n) = O(E \lg V)$

Prim's

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ φ
        u = Extract-MIN(Q)                  } O(V lg V)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u,v) < v.key     } O(lg V)
                v.π = u
                v.key = w(u,v)
```

$O(V \lg V + E \lg V) + O(E)$

$\Rightarrow O(E \lg V)$ ans

24-3

(6) Apply BELLMAN-FORD algorithm.

$R[i_1, i_2] \cdot R[i_2, i_3] \ldots R[i_k, i_1] > 1$

$\therefore \ln(R[i_1, i_2]) + \ln(R[i_2, i_3]) + \ldots + \ln(R[i_k, i_1]) > 0$

$-\ln(R[i_1, i_2]) - \ln(R[i_2, i_3]) \ldots - \ln[R[i_k, i_1]] < 0$

(b) To do this, we first perform the same modification of all the edge weights as done in part (a). Then apply the relax operation on all the edge at $|V|-1$ times.

Bellman-Ford recap

→ Shortest path estimate : $v \in V$, has a distance estimate $v.d$ which represents the shortest known distance from s to v.

Initially, $s.d = 0$ & $v.d = \infty$ $\forall s \neq v$

→ Relax up to $|V|-1$ times.

if $u.d + w(u,v) < v.d$
then $v.d = u.d + w(u,v)$.

→ If the algo. performs an additional pass over all edges to look for negative-weight cycles. If any edge $(u,v) \in E$ can still be relaxed, it indicates the presence of a negative weight cycle reachable from s.

24-3-10

(7) The proof of correctness of Dijkstra's Algorithm.

The key fact was that $\delta(s, y) \leq \delta(s, u)$. It is claimed that this holds because there are no negative edge weights, but in fact that is stronger than is needed. This always holds if y occurs on a shortest path from s to u & $y \neq s$ because all

edges on the path from y to u have non-negative weight. If any had negative weight, this would

$s \to y \to u$

imply that we had "gone back" to an edge incident with s, which implies that a cycle is involved in the path, which would only be the case if it were a negative-weight cycle. However, these are still forbidden.

24.3-8

(8) Step(1): Insert — $O(1)$

Step(2): Extractmin → $O(m+k)$

Step(3): DECREASE-key → $O(1)$

To apply this kind of min-priority queue to Dijkstra algo., we need to let $k = (|V|-1)W$ & We also need a seperate list for keys with values ∞

The no. of operations m is $O(V+E)$ [Insert, Extractmin, Decrease-key]

$$O(V+E+VW) = O(VW+E).$$

26.2-11

(9)

```
EDGE-CONNECTIVITY(G)
    k = ∞
    Select any vertex u ∈ G.V
    for each vertex v ∈ G.V - {u}
        set up the flow N/w G[u,v] ∞.
        find the maximum flow f[u,v] on G[u,v]
        k = min(k, |f[u,v]|)
    return k.
```

$G_{uv} = \min_{u \in V-\{u\}} \{|f_{uv}|\}$

k → minimum No. of edges.

22.4-5

(10)

```
TOPOLOGICAL-SORT(G)
    for each u ∈ G.Adj[v]
        u.indegree = u.indegree - 1
        if v.indegree == 0
            ENQUEUE(Q, v)
    for each vertex u ∈ G.V
        u.indegree = 0
    for each vertex u ∈ G.V
        for each v ∈ G.Adj[u]
            v.indegree = v.indegree + 1
    for each vertex u ∈ G.V
        if u.indegree ≠ 0
            report that there is a cycle.
    Q = ∅
    for each vertex u ∈ G.V
        if u.indegree == 0
            ENQUEUE(Q, u)
    while Q ≠ ∅
        u = DEQUEUE(Q)
        output u.
```

Lemma 22.11

(ii) Suppose that a DFS search produces a back edge (v, u). Then v is an ancestor of vertex u in the DFS forest. Thus, G contains a path from v to u & back edge (v, u) completes a cycle.

$\Leftarrow$: Suppose that G contains a cycle C. We show that a DFS of G yields a back edge.

Let u be the first vertex to be discovered in C & let (u, v) be the preceding edge in C.

→ At time $v.d$, the vertices of C form a path of white vertices from v to u.

→ By the white-path theorem, vertex u becomes a descendant of v in the depth first forest.

Therefore, (u, v) is a back edge.

modEXP

If $b = 0$, return 1 (since $a^0 = 1$)

If b is even, compute $(a^{b/2} \bmod p)^2$ & [illegible]

If b is odd, compute $(a \cdot (a^{b-1} \bmod p)) \bmod p$.

Euclid's [illegible]

while $b \neq 0$

$r = a \bmod b$

update $a = b$ & $b = r$

when $b = 0$ [illegible] Real principal (a, b)

$gcd(a, b) = 1$

$(\sim x_1)$ (x_2) (x_3)

(x_1)

$(\sim x_2)$

(x_3)

$\phi = C_1 \wedge C_2 \wedge C_3$

Assignment satisfies $C_1 \to \sim x_2$ & C_3 & $C_2 \to x_3$

$C_1 = x_1 \vee \sim x_2 \vee \sim x_3$

Clique [illegible] → (0)

$(a + b) \wedge (a + b)$

$a \vee b = (\ )^2$

$O((\log a + \log b)^2)$

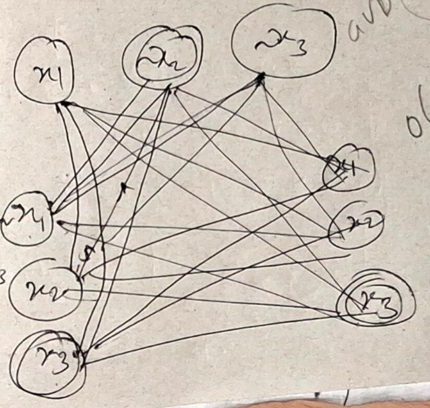
$\sim a + ob$

$\sim b + oa$

$x_1 = 0/1$

$x_2 = 0$

$x_3 = 1$



$C_2 = \sim x_1 \vee x_2 \vee x_3$

$C_3 = x_1 \vee \sim x_2 \vee x_3$

$O(\log$ [illegible]