09/01/19

```
c = 0;                                  (*)
for (i = 0; i < 1024; i = i * 2)
{
    c = c + 1;
}
printf ("%d", c);
```

So loop runs inf. no. of times

→ say, takes a const. time 'x' on a particular m/c.

(*) ⇒ if i = 1, then c = 9 ⇒ depends on how many times the loop is repeated

2^1
2^2
⋮
2^x

⇒ after x no. of iterations.

So, $2^x < 1024$

$\Rightarrow x < (\log_2 1024)$

Particularly useful when its 1024 is large i.e. 30456 etc.

CO

1) Analyze & compute the efficiency of algo in terms of running time using asymptotic analysis

2) Understand how to write the proof of correctness of algo

3) Be familiar with different algorithmic strategies such as brute-force, greedy, divide-and-conquer, backtracking, branch-&-bound, heuristics-based, etc.

4) Describe major algorithms, such as graph & tree algorithms, randomized algo, pattern matching algo, etc & analyze the worst case running time of these algo using asymptotic analysis

5) Synthesize efficient algo in common engineering design situations.

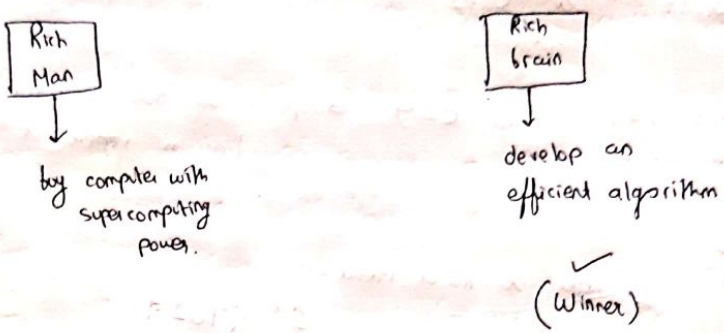
Scanned by CamScanner

Slide1 : Why study Algo?

→ Imp. for all other branches of CS

→ plays a key role in modern technological innovation

– "Everyone knows Moore's Law – a prediction made in 1965 by Intel...." ..."density of transistors in IC would continue to double every 1-2 yrs."



Eg: All Permutations of 100 numbers $\Rightarrow 2^{100}$

Say, 1 ns for a computation

So, 2^{100} ns = ↑↑ hrs

Even after considering Moore's Law & speedy hardware, 1st kind of probs can't be solved in efficent time

→ plays a key role in modern technological innovation

→ Provides novel "lens" on processes outside of CS & technology

→ challenging

→ fun

Eg: Prime No. checking → $O(\sqrt{n})$

Eg: Finding two prime factors of a no. → Complexity?

Scanned by CamScanner

At every step, the qs. should be put ⟹ "Can we do it better?"

Integer Multiplication

Input: 2 n-digit numbers x & y

Output: product $x * y$

"Primitive Operation" — add or multiply 2 single-digit numbers

(Efficient algo)
This becomes essential when there are say, 20-digit numbers

Eg:

```
  5678
x 1234
------
 22712
 17034 -
 11356 - -
 5678 - - -
------
 7006652
```

Roughly n-operations per row upto a const.

# of operations overall:
~ const. $* n^2$

A Recursive Algo

$x = 10^{n/2} a + b$ & $y = 10^{n/2} c + d$

a, b, c, d are n/2 digit no.

Eg: a = 56, b = 78, c = 12, d = 34

$\therefore x.y = (10^{n/2} a + b)(10^{n/2} c + d)$

$= 10^n ac + 10^{n/2}(ad + bc) + bd$ —— (1)

(4 products)

Recursively compute ac, ad, bc, bd & then compute (1) in the obvious way

$5678 = 10^2 . 56 + 78$

Scanned by CamScanner

After developing/designing this algo, we hv to do analysis of the algo to know whether it is better than the prev.

11/01/19

1.1 Algorithms

Algo. any well-defined computational procedure that takes some value/a set of values, as i/p & produces some value, or set of values as o/p

OR a method of solving a prob., using a sequence of well-defined steps

Eg: Sorting Pblm

I/P: a seq of n nos $\langle a_1, a_2, \ldots, a_n \rangle$

O/P: A permutation of the i/p sequence s.t

$a_1' \leq a_2' \leq \ldots \leq a_n'$

Every step shld be as well-defined as, a child can understnd & implement each step using his/her school-level maths

---

Sorting Problem

→ choose one permutation out of all possible permutations of n nos.

Step1 : Compute all permutations

Step 2 : For each permutation, P, check p is sorted or not
if p is sorted, goto step 3

Step 3 : Print sorted list

Scanned by CamScanner

Most of the sorting algo looks into the patterns of input.
Say, i/p is partially ordered

1. Selection sort
2. Bubble "
3. Insertion "
4. Quick "
5. Merge "
6. Heap "

Divide & Conquer → 4, 5

(CBSA)
Comparison based sorting algo.

Thm: CBSA can't hv a complexity $< O(n \log n)$

7. Counting "
8. Bucket "
9. Radix "

Comen slides:

Analysis of algo:

The theoritical study of comput-prog performance & resource usage.

More imp than paformance

→ modularity
→ correctness
→ maintainability
→ functionality
→ robustness

→ user-friendliness
→
→ robustness
→
→
→

Scanned by CamScanner

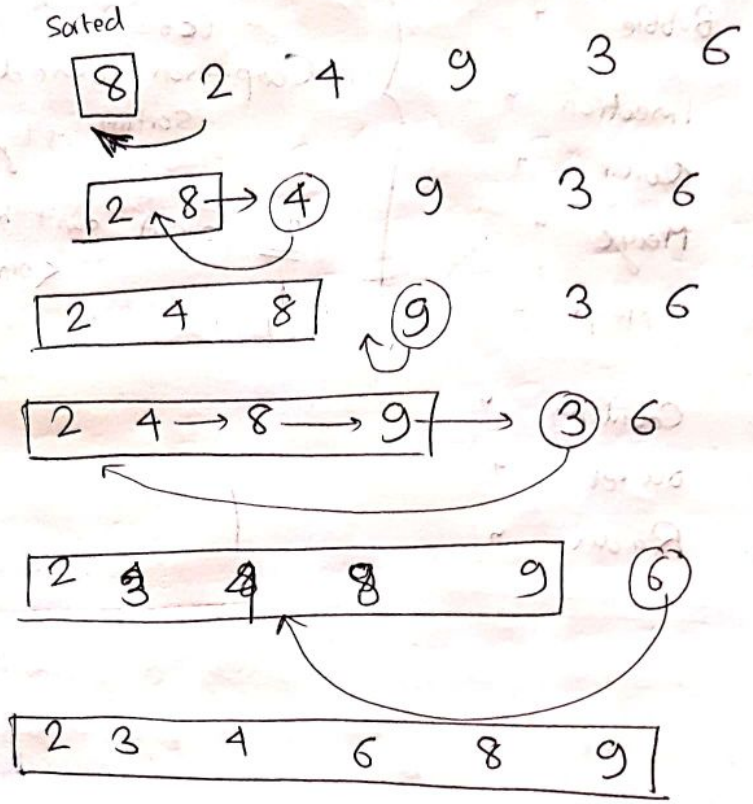
Analysis of Sorting Algo

(CLS slides)

16/01/19

Insertion Sort: (first algo)

INSERTION-SORT(A, n)



Running Time

→ Running time depends on i/p: an already sorted seq is easier to sort

→ Parameterize the running time by the size of the i/p, since short sequences are easier to sort than long ones

→ Generally we seek upper bounds on running time, bcos everybody likes a guarantee

Scanned by CamScanner

→ able to do basic computation

basic arithmetic operations (add^n, looping, branching, etc)

Kinds of Analysis

Worst case (usually)

$T(n)$ = max time of algo on any i/p of size n

Avg case (sometimes)

$T(n)$ = avg time of algo over all i/p of size n.

M/c independent time

Big Idea: Ignore m/c dependent const

Look at growth of $T(n)$ as $n \to \infty$

" Asymptotic Analysis "

```
                                                    Cost    times
for j ← 2 to n                                      c1      (n)
    // insert A[j] into sorted A[1...j-1] sequence  0       0
    do key ← A[j]                                   c2      (n-1)
       i ← j-1                                      c4      (n-1)
       while i>0 and A[i] > key                     c5      Σ_{j=2}^{n} t_j
          do A[i+1] ← A[i]                          c6      Σ_{j=2}^{n} (t_j - 1)
             i ← i-1                                c7      Σ_{j=2}^{n} (t_j - 1)
       A[i+1] ← key                                 c8      (n-1)
```

Scanned by CamScanner

→ c_1 will be executed $n-1+1$ times
(false checking last time)

→ for a particular j, inner loop executed t_j times

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

($T(n)$ ↓ total running time depends on the i/p size n)

When best case occurs : (when i/p array is sorted)

Innermost loop will be executed only once

So, $t_j = 1$ (∵ need to chck condition ie, A[i] > key & it will be false)

$$\therefore T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= an + b \Rightarrow \text{Linear in i/p size}$$

where a & b are constants

When worst case occurs : (when i/p is sorted in reverse order)

Here, $t_j = j$ (In worst case, innermost loop will be executed j times
⇓
j-1 shifting operation
+
1 checking to make the loop false)

Scanned by CamScanner

$$\therefore T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8 (n-1)$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$= c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \left(\frac{n(n+1)}{2} - 1\right) + c_6 \left(\frac{n(n-1)}{2}\right) + c_7 \left(\frac{n(n-1)}{2}\right) + c_8 (n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - \left(c_2 + c_4 + c_5 + c_8\right)$$

$$= an^2 + bn + c$$ (quadratic in nature)

When avg case occurs ? depends on the distribution of all possible inputs

Scanned by CamScanner

(XEROX) 21/01/19

Average case analysis of Insertion Sort (CRAMER, 1750)

Assume: that every one of $n!$ different permutations of the n items are equally likely as input

A permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of $1 \ldots n$ is simply a list of numbers between 1 and n in some order.

(π_i, π_j) is called inversion if $i < j$, but $\pi_i > \pi_j$

Eg: $\pi = [3\ 5\ 1\ 4\ 2]$ (one of the inversion)

has inversion $(1,3), (1,5), (2,3), (2,4), (2,5), (4,5)$

Say for eg:

π is already in ascending order.

Eg: $\pi = [1, 2, \ldots, 5]$ $\Rightarrow$ it has no inversions

for eg: π is in reverse order

$\pi = [5, 4\ 3\ 2\ 1]$ It has 10 inversions

max no. of inversions $\leftarrow \binom{n}{2}$

The goal of sorting is to remove inversion.

(bcos our objective is to go from max. no. of inversion to 0 no. of inversions)

For $\pi = (5\ 4\ 3\ 2\ 1)$

if this 3 & 4 is swapped, one iteration is used.

We have, $\pi = (5\ 3\ 4\ 2\ 1)$

So, Swapping (not a part of Insertion sort) an adjacent pair of positions that are out of order, decreases the no. of inversions by exactly 1

But for analysis of our algo we assume that,

Shifting right & inserting an element into the sorted sub-list is actually removing swaps. (or inversion)

Eg: 2 3 1 (2 inversions)

1 2 3 (2 inversions are removed)

Total no. of swaps performed by Insertion sort

= no. of inversions present in the input

So, $I_{i,j} = \begin{cases} 1 & \text{if } (\pi_i, \pi_j) \text{ is an inversion} \\ & (\text{means } i<j \wedge \pi_i > \pi_j) \\ 0 & \text{if not inversion} \end{cases}$

$I_{i,j}$ → a Boolean variable

Scanned by CamScanner

Total no. of inversions / swaps,

$$I = \sum_{\substack{\forall i \, \forall j \\ i<j}} I_{i,j}$$

Now, the ques is, what is the expected no. of inversions for any possible i/p randomly chosen from n! different permutations of n items?

$E(I)$ → Expected value of I.

$$= E\left(\left[\sum_{i<j} I_{i,j}\right]\right), \quad \forall i \; \forall j.$$

$$= \sum_{i<j} E(I_{i,j}), \quad \forall i, \forall j \qquad (\star)$$

Next ques, What is the value of $E(I_{i,j})$?

→ For any fixed (π_i, π_j), it is equally likely that, in a random permutation, π_i will appear before π_j and π_i will appear after π_j.

Thus,

$$\text{prob}(I_{i,j} = 1) = \text{prob}(I_{i,j} = 0) = \frac{1}{2}$$

Eg: 1, 2, 3 → 3 nos. = 6 possible permutations

1, 2, 3	prob $(I_{1,2} = 1) = 3/6$
1, 3, 2	prob $(I_{1,2} = 0) = 3/6$
2, 1, 3	So prob. $= 1/2$
2, 3, 1	
3, 1, 2	
3, 2, 1	

Scanned by CamScanner

Expected value of random variable = Σ prob.

So,

$$E(I_{i,j}) = 1 \cdot \text{prob}(I_{i,j} = 1) + 0 \cdot \text{prob}(I_{i,j} = 0)$$

$$= 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{2}$$

$$= 1/2$$

1 2 3

1 (2 . . . . n) n-1

2 (3 . . . n n-2

So, from (*),

$$E = \sum_{\substack{\forall i, \forall j \\ i<j}} E(I_{i,j})$$

$$= \sum_{\substack{\forall i, \forall j \\ i<j}} \frac{1}{2} \qquad = \frac{1}{2} \cdot \sum_{\substack{\forall i \forall j \\ i<j}} 1$$

$$= \frac{n(n-1)}{4}$$

$\frac{n(n-1)}{2}$

If the i/p is sorted in reverse order.

Total no. of inversions $= \binom{n}{2} = \frac{n(n-1)}{2}$

∴ The avg. case running time of insertion sort half of the worst case running time

Algorithm Design :

Important issue is the Programmer's Tool Box

↓

try tools available in the toolbox to solve the problem.

Scanned by CamScanner

Important Paradigms

(1) → Divide & Conquer
(2) → Greedy Approach
(3) → Dynamic Programming Approach
(4) → Approximate Algorithms

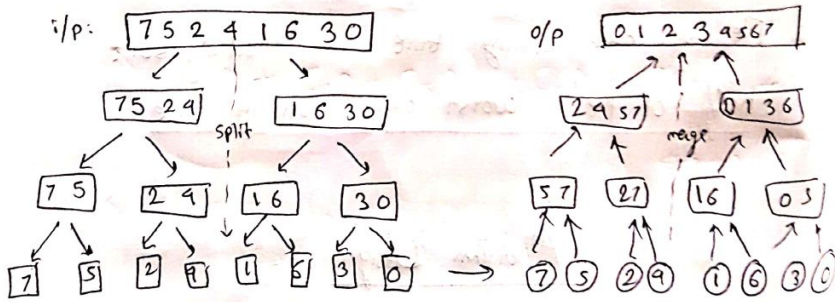
(slide/notes)

Summarizing the main elements to a divide-and-conquer solution are:

- Divide (the problem into a smaller no. of pieces)
- Conquer (solve each piece by applying divide-and-conquer recursively to it), etc...
- Combine (the pieces together into a global solution)

Recurrence relations

Merge Sort



arrow is necessary bcs its actually a Recursive call

Figure 4: MergeSort

Scanned by CamScanner

In each recursive call, input size is divided by 2.

Algo

```
MergeSort (array A, int p, int r) {        // p: lower bound of the arr; r: upper bound of the arr.
    if (p<r) {                             ---- c1
        (divide)  q = (p+r)/2              ---- c2
        T(⌊n/2⌋)  MergeSort (A, p, q)      -> i/p size is p to q
        T(⌈n/2⌉)  MergeSort (A, q+1, r)    -> i/p size is (q+1) to r
        (combine) Merge (A, p, q, r)       ---- kn
    }
}
```

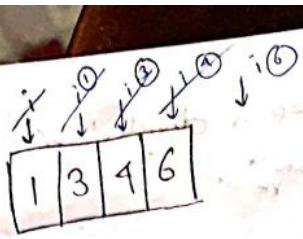
max^m running time is needed for this procedure

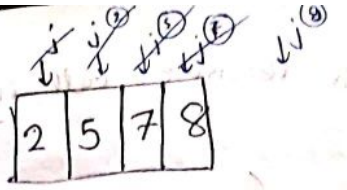
25/01/19

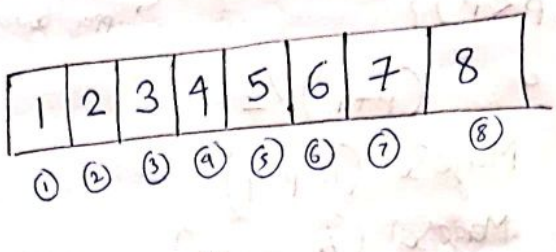
```
Merge (array A, int p, int q, int r) {
    array B [p..r]
    i = k = p          ---- c1
    j = q+1            ---- c2
    while (i<=q and j<=r) {                ---- c3
        if (A[i] <= A[j]) B[k++] = A[i++]
        else              B[k++] = A[j++]      } - c4
    }
```

3 while loops together can execute at most n-times (where

Scanned by CamScanner







Supposing the 3 while-loops execute n_1, n_2 & n_3 times,

$$T(n) = c_1 + c_2 + c_3 n_1 + (n_1 - 1)c_4 + c_5 n_2 + c_6 n_3 + c_7 n$$

$$= c_1 + c_2 + c_3 n_1 + c_4 n_1 - c_4 + c_5 n_2 + c_6 n_3 + c_7 n$$

$$T(n) = (c_3 n_1 + c_4 n_1 + c_5 n_2 + c_6 n_3) + c_7 n + c_1 + c_2 - c_4$$

$$\leq (c_3 n + c_4 n + c_5 n) + c_7 n + c_1 + c_2 - c_4 \quad (\because n_1 + n_2 + n_3 = n)$$

$$\leq (c_3 + c_4 + c_5 + c_7) n + c_1 + c_2 - c_4$$

$$\leq an + b$$

So the upper bound of this merge procedure is linear in i/p size.

Scanned by CamScanner

For convenience of computation, we say,

$$T(n) = k.n$$

↳ also a constant.

running time for Merge Sort

$T(n)$. (where n is the i/p size)

So, our orig func^n (★)

recurrence relation

$$T(n) = c_1 + c_2 + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + kn$$

if $n=1$, $T(1) = 1$

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + cn$$

$$\leq 2T(\lceil n/2 \rceil) + cn$$

we are taking ceiling always, so can simply remove that

$$\leq 2\left(2.T(n/4) + c.n/2\right) + cn$$

$$\leq 2^2 T(n/2^2) + cn + cn$$

$$\leq 2^2 T(n/2^2) + 2cn$$

⋮

At step x,

$$T(n) \leq 2^x . T\left(\frac{n}{2^x}\right) + x.cn \quad (★_2)$$

Say at step x, $T(n/2^x) = T(1)$

$$\Rightarrow \frac{n}{2^x} = 1$$

$$\Rightarrow \boxed{x = \log_2 n} \quad (★_1)$$

Scanned by CamScanner

From (*1) & (*2)

$$T(n) \leq 2^{\log_2 n} T(1) + \log_2 n \, cn$$

$$\leq n + cn \log_2 n$$

(upper bound of the running time of this procedure)

So,

Insertion Sort

$$T(n) = an^2 + bn + c$$

Merge Sort

$$T(n) = n + cn \log_2 n$$

value of a, b, c depends on what type of computer, this algo is actually run

Scanned by CamScanner

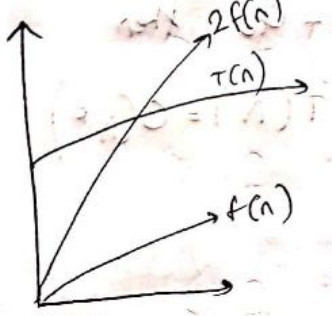
Suppose :

n	$n^2 + 2n$	$n + 100 n \log_2 n$
2	8 ✓	202
4	24 ✓	804
⋮	⋮	
1024	$1024^2 + 2 \times 1024$ $\simeq 10^6$	$\simeq 10^6$
2048	$\simeq 4 \times 10^6$	$\simeq 10^6$ ✓

✓ ≡ better

We are actually doing the asymptotic analysis of the algo.

Big-Oh: Formal Def^n



Formal def^n: $T(n) = O(f(n))$ if and only if there exist constants $c, n_0 > 0$ s.t.

$$T(n) \leq c \cdot f(n)$$

for all $n \geq n_0$

Warning: c, n_0 can't depend on n

Scanned by CamScanner

Eg: $T(n) = 10n^2 + 2n + 3$

Prove $T(n) = O(n^2)$ → to prove we have to find some n_0, $c > 0$, & the condition

A. $T(n) = 10n^2 + 2n + 3$

$\leq 10n^2 + 2n^2 + 3n^2$

$\leq (15)n^2$

$n_0 = 1$
$c = 15$

I

Eg #1

Claim: if $T(n) = a_k n^k + \dots + a_1 n + a_0$ then

$T(n) = O(n^k)$

Proof: Choose $n_0 = 1$ &

$c = |a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_0|$

Need to show that: $\forall n \geq 1, T(n) \leq c \cdot n^k$

We have, for every $n \geq 1$

$T(n) \leq |a_k| n^k + \dots + |a_1| n^1 + |a_0|$

$\leq |a_k| n^k + \dots + |a_1| n^k + |a_0| n^k$

$= c \cdot n^k$

Scanned by CamScanner

NOTE: Ignore the lower Order terms, take the most dominating term (highest order term)

$$T(n) = O(n^2) = O(n^3) = O(n^4)$$

Eg: $T(n) = 10n^2 + 2n + 3$

$\leq 10n^3 + 2n^3 + 3n^3$

$\leq 15n^3$

$= O(n^3)$ also true

But we do

$\leq 10n^2 + 2n^2 + 3n^2$

$\leq 15n^2$

$= O(n^2)$

} Tight Upper Bound

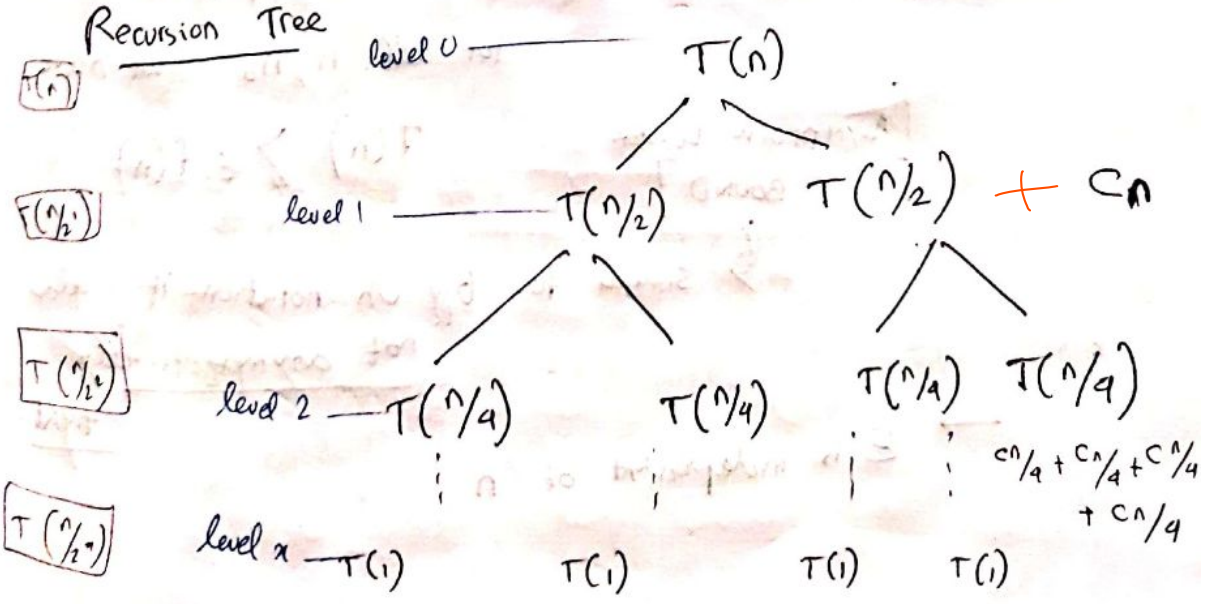
Whenever we compute Big-O notation we'll try to compute

01/02/19

$$T(n) \leq 2T(n/2) + cn$$

$T(1) = 1$

Recursion Tree



Scanned by CamScanner

Finally,

$$T(n) = \sum_{1}^{x} cn$$

i.e. $cn + cn + \cdots + cn$ (x - times)

n is reducing by 2 in each step

$$n/2^x = 1$$

$$\Rightarrow x = \log_2 n$$

So, $= cn \log_2 n$

Asymptotic Order

(we compare 2 algos by this notation)

Omega Notation : Ω

Def^n : $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$, so that for all $n > n_0$, we have

$$T(n) \geq \varepsilon . f(n)$$

[ASYMPTOTIC LOWER BOUND]

↳ Similar to Big-Oh notation, it is also not asymptotically tight.

→ ε is independent of n

Scanned by CamScanner

Q Check whether $T(n) = pn^2 + qn + r = \Omega(n^2)$?

A: $T(n) = pn^2 + qn + r$

$\geq pn^2$ (if p, q, r are all +ve constants)

If $n_0 = 1$, $\forall\ n \geq n_0$ this cond^n holds.

So according to defn of Ω,

$= \Omega(n^2)$ $\boxed{\varepsilon = p}$

Theta Notation : Θ → This is asymptotically tight bound (bcos from below & upper we hv same kind of funcn)

Defn: $T(n) = \Theta(f(n))$ if

$T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$

Eg: $T(n) = pn^2 + qn + r$

Here, $T(n) = O(n^2)$ and also $T(n) = \Omega(n^2)$

$\therefore T(n) = \Theta(n^2)$

Actual Idea: $T(n)$ grows exactly like $f(n)$ to within a constant factor.

Scanned by CamScanner

Q Show that: $(\ln n)^k = O(n^\alpha)$

A. Hv to use LIMIT METHOD

Limit Method

Defn of Big-Oh using Limit Method: $T(n) = O(f(n))$ if

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = c < \infty$$

(★) including the case when it is 0

Intuition: $T(n)$ becomes insignificant relative to $f(n)$ or its ratio converges to positive const. as $n \to \infty$

Say given, $T(n) = 4n^3 + 10n^2 + 5n + 1$

Show that,

$$T(n) = O(n^4)$$

$f(n) = n^4$

$$\lim_{n \to \infty} \left(\frac{4n^3 + 10n^2 + 5n + 1}{n^4} \right) = \lim_{n \to \infty} \left(\frac{4}{n} + \frac{10}{n^2} + \frac{5}{n^3} + \frac{1}{n^4} \right)$$

$= 0$ (satisfies ★)

Hence, proved that $T(n) = O(n^4)$

Scanned by CamScanner

Defn of Big-Ω using Limit Method :

$T(n) = \Omega(f(n))$ if

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = c > 0$$

including the case, where limit is ∞

Intuition: $T(n)$ becomes arbitrarily large relative to $f(n)$ or this ratio converges to a positive constant as n goes to ∞ (infinity)

Defn. of Θ using Limit Method :

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

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = c$$

where c is some constant such that $0 < c < \infty$ (not 0, not ∞)

Intuition: The ratio of $T(n)$ and $f(n)$ converges to a positive constant as $n \to \infty$

So, if $T(n) = \Theta(f(n))$

$$\Rightarrow \begin{cases} T(n) \leq c_1 f(n) \\ T(n) \geq c_2 f(n) \end{cases}$$

Definitely $c_1 > c_2$

bcos $c_2 f(n) \leq T(n) \leq c_1 f(n)$

$$c_2 \leq \frac{T(n)}{f(n)} \leq c_1$$

The ratio of $T(n)$ & $f(n)$ lies betⁿ c_1 & c_2

⇒ so always a +ve const betⁿ 0 & ∞

Scanned by CamScanner

Eg: $T(n) = 4n^3 + 10n^2 + 5n + 1$

$f(n) = n^2$

Show that : $T(n) = \Omega(f(n))$

$$\lim_{n \to \infty} \left(\frac{4n^3 + 10n^2 + 5n + 1}{n^2} \right) = \infty$$

Hence, $T(n) = \Omega(n^2)$

Eg: $T(n) = pn^2 + qn + r$ $\quad T(n) = O(n^2)$

$f(n)$

A. $$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right)$$

$$= \lim_{n \to \infty} \frac{pn^2 + qn + r}{n^2}$$

$$= \lim_{n \to \infty} \left(p + \frac{q}{n} + \frac{r}{n} \right)$$

$= p$

Hence proved that.

$T(n) = O(n^2)$

Relation bet^n O, Ω, Θ (represented by Venn diag)

Scanned by CamScanner

Little -Oh : $T(n) = o(f(n))$ if

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = 0$$

Little Omega : $T(n) = \omega(f(n))$ if

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty$$

Q For all $k > 0$, show that $(\ln n)^k = O(n^{\alpha})$ (alpha)

A: $$\lim_{n \to \infty} \frac{(\ln n)^k}{n^{\alpha}}$$

Using L'Hospital rule, $\left[\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = \lim_{n \to \infty} \frac{T'(n)}{f'(n)} \right]$

H.W.

Scanned by CamScanner

→ Always look for algebraic simplifications

→ You must give rigorous proof

→ Using limit method (if possible) is the best

→ Apply L' Hospital's rule if need be

→ Give as simple (& tight) expressions as possible

Divide & Conquer Algo :

Master Theorem

$$T(n) = aT(n/b) + O(n^d)$$

for some constants $a \geq 1$, b, $b > 1$

and $d \geq 0$

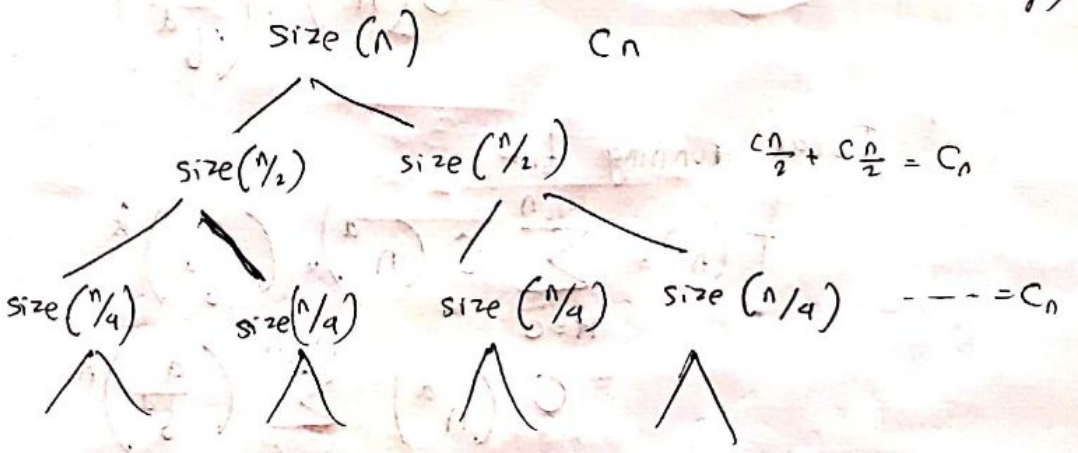
$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log_b n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

(any base)

how to check relationship bet^n d, b, a

Eg: $a = 2, b = 2, d = 1$ i.e. $T(n) = 2T(n/2) + O(n)$

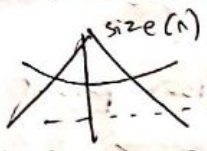
(recurrence eqn. for Merge Sort algo)



Scanned by CamScanner

General Case

level 0 → cn^d



branching factor a

level 1 → $c\left(\frac{n}{b}\right)^d + c\left(\frac{n}{b}\right)^d + \cdots + \text{upto } a \text{ terms}$

$= a \cdot c \cdot \left(\frac{n}{b}\right)^d$

size (n/b) size (n/b) -- size (n/b)

level 2 → $a^2 \cdot c\left(\frac{n}{b^2}\right)^d$

⋮

At level k → $a^k \cdot c \cdot \left(\frac{n}{b^k}\right)^d$

⋮

At level x, say, $\frac{n}{b^x} = 1$

$\Rightarrow \boxed{x = \log_b n}$

∴ At level k, $a^k \cdot c \cdot \left(\frac{n}{b^k}\right)^d$

$= a^k \cdot O\left(\left(\frac{n}{b^k}\right)^d\right)$

$= O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$

∴ Total running time,

$$T(n) = \sum_{k=0}^{\log_b n} O(n^d) \cdot \left(\frac{a}{b^d}\right)^k$$

$$= O(n^d) \cdot \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k$$

$$= O(n^d) \cdot \sum_{k=0}^{\log_b n} r^k$$

where $r = \left(\frac{a}{b^d}\right)$

Scanned by CamScanner

Since we are doing asymptotic analysis, we have to get either the highest or lowest term of the sequence, based on the value of $r = \left(\frac{a}{b^d}\right)$

{ based on = 1, < 1, > 1

Case (1) ratio is < 1 $(r < 1)$:

Then the series is decreasing and its sum is just given by the first term (∵ highest term)

first term $= \left(\frac{a}{b^d}\right)^0 = 1$

$\therefore \boxed{T(n) = O(n^d) \cdot 1 = O(n^d)}$

Now, $r < 1$

$\Rightarrow \frac{a}{b^d} < 1 \Rightarrow \boxed{d > \log_b a}$

Case (3) the ratio is > 1 $(r > 1)$:

$\Rightarrow \frac{a}{b^d} > 1 \Rightarrow \boxed{d < \log_b a}$

The series is increasing and its sum is given by the last term

$\therefore T(n) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right)$

$= n^d \left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right)$

$= n^d \left(\frac{a^{\log_b n}}{n^d}\right) = a^{\log_b n}$

$= a$

Scanned by CamScanner

$$= a^{(\log_a n).(\log_b a)}$$

$$= a^{\log_b d}$$

$$= n$$

$\log_b n = \log_a n \times \log_b a$

Case (2) The ratio = 1 $(r = 1)$

$$\frac{a}{b^d} = 1 \Rightarrow \boxed{d = \log_b a}$$

$\log_b n$

In this case, all n terms of the series are equal to $O(n^d)$

$$\therefore T(n) = O(n^d \log_b n)$$

$$= O(n^d \log n)$$

$\because \log_b n = \frac{\log n}{\log b}$

$\Rightarrow O(\log_b n) = O(\log n)$

Integer Multiplication

$$\begin{array}{r} 12 \\ \times\ 13 \\ \hline 36 \\ 12\ x \\ \hline \end{array}$$

$O(n^2)$

mult / additions bcos 1 mult for each digit

$\therefore T(n) = O(n^2)$ on multiplying 2 n-digit no.

Scanned by CamScanner

Say we hv 2 nos. x, y

$x = (x_1, x_0)$ — $n/2$ bits each

$y = (y_1, y_0)$ — $n/2$ bits each

$\therefore x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

Both x & y are represented in binary form

$\therefore xy = x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0$

(2^n: order of (n) (∵ we hv to shift n times) → bring in n 0's to the right)

$T(n)$ ← xy; $T(n/2)$ ← $x_1 y_1$, $x_1 y_0$, $x_0 y_1$, $x_0 y_0$

(∵ prob size becoming half)

Eg: $4567 = 45 \times 10^2 + 67$

$9835 = 98 \times 10^2 + 35$

Thus combining sol^n takes $O(n)$

$\therefore T(n) \leq 4T(n/2) + O(n)$

By Master Thm

$d = 1 < \log_b a = \log_2 4 = 2$ (3rd case)

$\therefore T(n) = O(n^{\log_b a}) = O(n^2)$

No gain than original (*)

Scanned by CamScanner

Another modification

Recursive - multiply (x, y):

write $x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

Compute $x_1 + x_0$ and $y + y_0$

P = recursive_multiply $(x_1 + x_0, y_1 + y_0)$

$x_1 y_1$ = Recursive_multiply (x_1, y_1)

$x_0 y_0$ = Recursive_multiply (x_0, y_0)

Return $x_1 y_1 \cdot 2^n + (P - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

$\therefore T(n) \leq 3T(n/2) + O(n)$

By M.T. Case 3 $\log_2 3 > 1$

$\therefore O(n^{\log_2 3})$

$= O(n^{1.58})$

Tim Roughgarden Slide ↓

Quick Sort : (Overview) 08/02/19

→ "greatest hit" algorithm

→ $O(n \log n)$ time "on average" works in place

i.e. minimal extra memory needed

Scanned by CamScanner

The Sorting Problem

I/p: array of n numbers, unsorted

3 8 2 5 1 4 7 6

o/p: Same nos. sorted in Incr. order.

Assumption: all nos. are distinct.

Partitioning Around a Pivot

Key Idea: Partition array around a pivot elem

→ Pick elem of array

Rearrange array so that:

left of pivot ⇒ less than pivot

right " " ⇒ greater " "

∴ puts pivot in its "rightful pos"

→ Linear O(n) time, no extra mem

→ Reduces prob size (Divide & Conquer)

Scanned by CamScanner

Randomized Q.S ⇒ randomly choose one of the elements as pivot

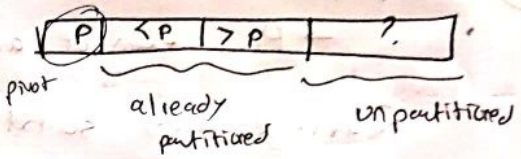
Then what will be the complexity

In-place Implementation

Assume: pivot = 1st elem of array

(if not, swap (pivot, 1st elem) as preprocessing step)

High Level Idea



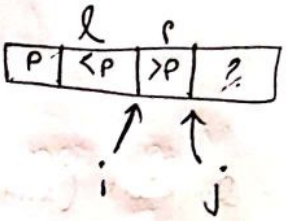
- Single scan thru array

Invariant: Everything looked at so far is partitioned

Partition Eg:

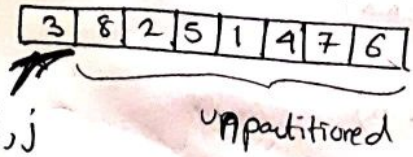
j: points to first elem of unpartitioned

i: points to middle of left & right partition

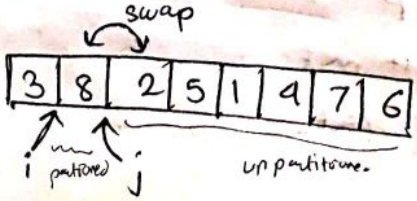


Initially

(1)



(2)



8 should belong to r. So we don't increment i, we incr j

(3) Since 2

3

(4)

fast step:

finally

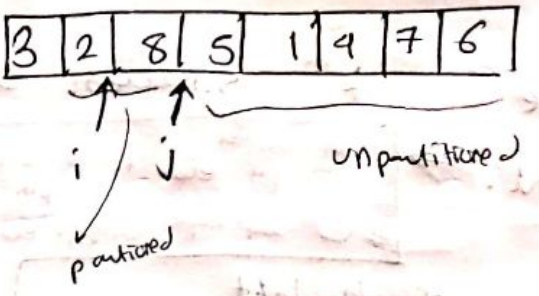
Complexity of this:

O(n)

Scanned by CamScanner

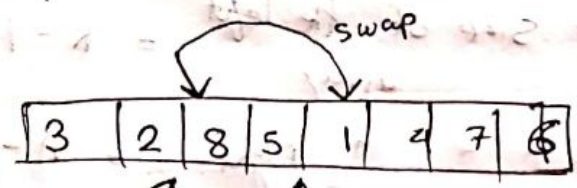
So now, no elem in l but 1 elem in r.

③ Since 2 should be in l,
we swap 2 & 8, while we incr i.

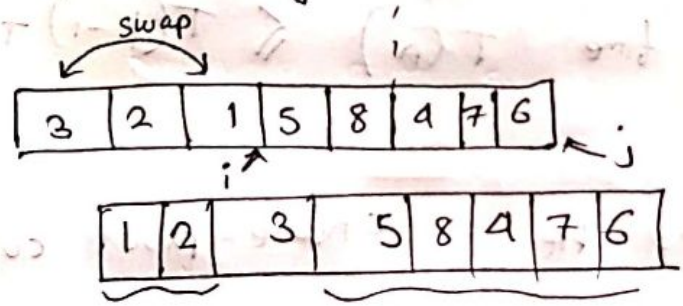


j will always be incr.
Whether i will be incr, will depend on swap

④



swap to update l & bring 8 in r



last step:

finally

1 2 3 5 8 4 7 6

< 3 > 3

(l & r taken as i/p bcos it chng in different recursive calls)

Pseudocode for Partn

- complexity: O(n)

how to give line by line expl. in exam

```
Partition (A, l, r)        → A [l....r]
    p = A[l]
    i = l + 1
    for j = l+1 to r       → if this is n times
                           → n-1 times
        if A[j] < p        → // if A[j] > p, do nothing
            swap A[j] and A[i]
            i = i + 1
    swap A[l] and A[i-1]
```

complexity of this: O(n)

Scanned by CamScanner

Correctness of the algo (To be covered in class)

↑ slide

Analysis of QuickSort Algo

→ The performance of quick sort depends on which element is chosen as pivot

→ Assume that we choose k^{th} smallest element as the pivot element. ⇒

Can consider:
① 1st elem is definitely some k^{th} smallest elem
OR ② swap k^{th} smallest elem (from ... with ...

$|L|$ = size of the left array $= k-1$

$|R|$ = " " " right " $= n-k$

→ Total Running time: $T(n) \leq T(k-1) + T(n-k) + O(n)$

(How to write these lines in analysis)

- We have designed the choose-pivot condition in such a way that it picks pivot, where $k = n/2$

(Its possible bcos, there exists an algorithm to find median in linear time)

Then,

$$T(n) \leq O(n) + 2T(n/2)$$

and $T(n) = O(n \log n)$ (as per Master Theorem)

Worst Case Analysis:

This particular situation occurs when array is sorted in ascending order.

Scanned by CamScanner

Analysis of QuickSort Algo

→ The performance of quick sort depends on which element is chosen as pivot.

→ Assume that we choose k^{th} smallest element as the pivot element. ⇒

Can consider:
① 1st elem is definitely some k^{th} smallest elem
OR ② some k^{th} smallest elem (found by any a/p with 1st elem

$|L|$ = size of the left array $= k-1$

$|R|$ = " " " right " $= n-k$

→ Total Running time: $T(n) \leq T(k-1) + T(n-k) + O(n)$

- We have designed the choose-pivot condition in such a way that it picks pivot, where $k = n/2$

(Its possible bcos, there exists an algorithm to find median in linear time)

Then,

$$T(n) \leq O(n) + 2T(n/2)$$

and $T(n) = O(n \log n)$ (as per Master Theorem)

Worst Case Analysis:

This particular situation occurs when array is sorted in ascending order.

Scanned by CamScanner

Then, $T(n) \leq O(n) + T(n-1)$ ← every time right array is very longer than left array

$\Rightarrow T(n) \leq cn + T(n-1)$

$\leq c(n+n-1) + T(n-2)$

$\leq c(n+n-1+n-2+\ldots+T(1))$

$= O(n^2)$

One of the drawbacks of Q.S. is that, worst-case sitn is very bad

But its less likely to occur bcos when we are supplying an array to be sorted, it is less possible for it to be sorted beforehandedly.

Average Case Analysis of QuickSort Algorithm : 13/02/19

Randomized version of QuickSort Algo:

→ Try to compute the expected running time of an i/p sequence. (for this we modify the partition func^n as a randomized partition algo)

```
int Rpartition (A, l, r)
    P = RANDOM (l, r)
    swap (l, p)   // swap pth element with lowest elemt
    return partition (A, l, r)
                (prev part^n algo)
```

pth elem is the

Scanned by CamScanner

We are choosing m randomly

→ it is uniformly distributed

∴ The probability of each $m \in [0, n-1]$ is $\frac{1}{n}$.

- Say, the avg case running time / expected running time of QuickSort algo for an input instance of size n

$$\therefore T(n) = O(n) + \left(\frac{1}{n}\sum_{m=0}^{n-1}\left(T(m) + T(n-m-1)\right)\right)$$

(∵ prob of occurrence of one such event = $1/n$)

$$\leq cn + \frac{1}{n}\sum_{m=0}^{n-1}\left(T(m) + T(n-m-1)\right)$$

So expected value

$$\leq n + \frac{1}{n}\sum_{m=0}^{n-1}\left(T(m) + T(n-m-1)\right)$$

For co... of c... deri... we co... $c=1$

∴ The upper bound of $T(n)$:

$$T(n) = n + \frac{1}{n}\sum_{m=0}^{n-1}\left(T(m) + T(n-m-1)\right)$$

$$n \cdot T(n) = n^2 + 2\sum_{i=0}^{n-1} T(i) \quad \text{——} (1)$$

$$\Rightarrow (n-1)T(n-1) = (n-1)^2 + 2\sum_{i=0}^{n-1-1} T(i) \quad \text{——} (2)$$

(∵ $T(0) + T(1) + \cdots + T(n-1) + T(n-2) + \cdots + T(0) = 2\sum_{i=0}^{n-1} T(i)$)

Subtracting (2) from (1),

$$n \cdot T(n) - (n-1) \cdot T(n-1) = n^2 + 2\sum_{i=0}^{n-1} T(i) - \left[(n-1)^2 + 2\sum_{i=0}^{n-2} T(i)\right]$$

Scanned by CamScanner

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

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

$$\Rightarrow n\,T(n) = (n+1)\,T(n-1) + 2n - 1$$

Dividing both sides by $(n+1)$:

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

Dividing by n,

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \quad \text{---(3)}$$

$\Rightarrow$ This is the recurrence eqn

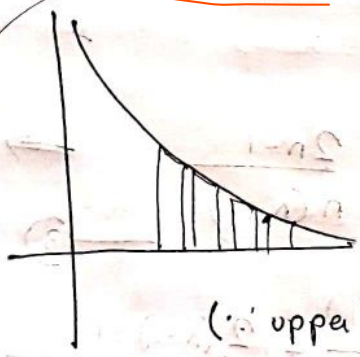
Scanned by CamScanner

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

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

Now,

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_{1}^{n+1} \frac{dx}{x}$$



(∵ upper bounded by that curve ↳ $\frac{1}{x}$)

$$= \ln(n+1)$$

$$\sum_{i=1}^{n} \frac{1}{i(i+1)}$$

$$= \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right)$$

(Telescopic series)

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

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

$$\therefore \frac{T(n)}{n+1} < 2\ln(n+1) - \left[1 - \frac{1}{n+1}\right]$$

(∵ +ve, so ignored)

$$\Rightarrow T(n) < 2(n+1)\ln(n+1)$$

$$\therefore T(n) < 2(n+1)\ln(n+1)$$

∴ Ave

So

Scanned by CamScanner

$$= \frac{2}{\log_2 e}(n+1)\log_2(n+1)$$

$$= 1.386..(n+1)\log_2(n+1)$$

∴ Average case performance is approximately 38% slower than the best case performance of quicksort algo

So randomized version of Q.S is actually implemented for any implementation of Q.S

Scanned by CamScanner

Graph Search Using **BFS** :

20/02/19

[all nodes initially unexplored]

- Mark S as explored

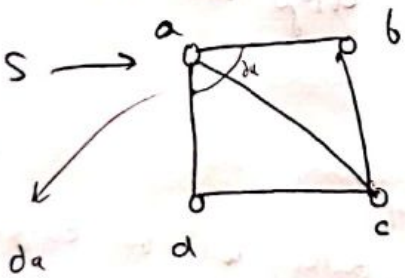
Let Q = queue data structure (FIFO) initialized with S

(*) while (Q ≠ 0) ⟶ O(n) (∵ it will run no. of vertices) no. of times)

- remove the first node of Q, call it V
- For each edge (v, w) (*)
  - if w unexplored
    - Mark w as explored
    - add w to Q (at the end)

In total it runs:
$\sum d_i$
$= 2m$
$= O(m)$
irrespective of how many times while loop executes

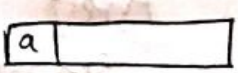
Eg: (for a connected graph)

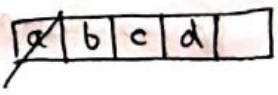


Say d_a = degree of a

⇓

Node a will contribute d_a no. of run of (*).

Q: 

Q: 

Roughly every node will contribute d_i no. of times. So (*)

∴ Total running time $= O(m+n)$

Eg.

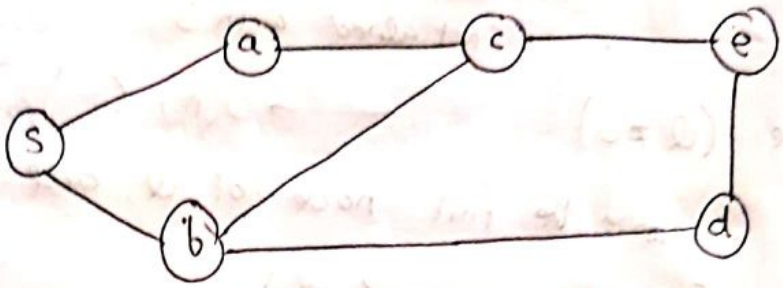
line 1 ⟶ O(n)
line 2 ⟶ O(m)
line 3 ⟶ O(n)

Scanned by CamScanner

Without using queue : $O(mn)$

Q Given a graph :

find the shortest path from a source to all possible nodes in the graph.



Brute - Force Algo :

Compute all possible s/n's & choose the optimal

→ Compute all possible paths from a source to the destination.

Apply minimum finding algorithm to choose the shortest path.

→ say you are taking a very large graph

So to compute all possible paths will be computationally expensive

Eg:

	a	b	c	d	
→	0	0	0	0	x
	0	0	0	1	x
	0	0	1	0	x
			⋮		
	1	1	1	1	

for a path, each node can either be present in it / not present

2^n paths

(out of these some valid some not (some) cases a & d must be 1 on the path)

Eg:

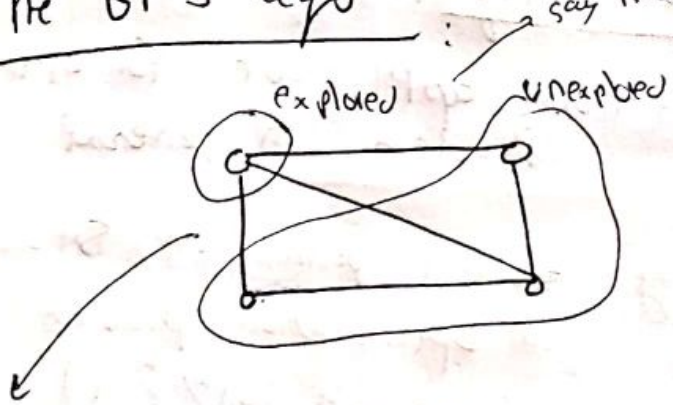
$O(2^n)$ ⇒ exponential

↓

Computationally intractable problem

Scanned by CamScanner

Using the BFS algo :



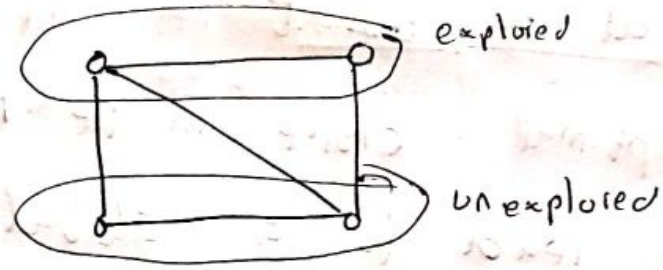
In each iteration, 2 partitions of the graph

⇓

explored & unexplored

3 routes a crossing the river

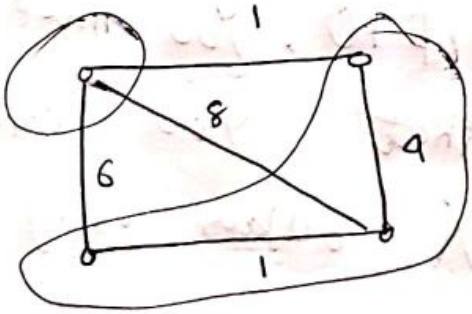
⇓



But this is only applicable when the graph is unweighted.

In practical appl^n, graphs are seldom unweighted.

If graph is weighted, BFS algo of no use



BFS only checks no. of paths that cut the river.

DIJKSTRA'S SHORTEST PATH ALGO. (Greedy algo)

Scanned by CamScanner

Greedy Algo:

→ more applicable for optimization Problem in general

→ finding shortest path from S to D is an optimization problem. (path with min^{m} cost)

→ A greedy algorithm always makes the choice that looks best at the ~~moment~~, i.e. it makes a locally optimal choice in the hope that this choice will lead to a global optimal solution.

Eg: **Coin Changing Problem**

In a certain country, the coin denominations are $1, $2, $5 and $10. You have to design an algorithm such that you can make change of any X dollars using the fewest no. of coins

Brute-Force Sol^{n}

C_1	C_2	C_3	C_4	
0	0	0	0	→ check whether = sum
0	0	0	1	→ " " "
0	0	1	0	→ " "

Find which are equal to sum? Amongst them which use lowest no. of coins? (i.e. smallest no of 1)

So, $O(2^n)$

Scanned by CamScanner

For any kind of greedy algo, there's an evaluation func^n ↓ (optimisation objective) based on that we'll decide which is your best

Eg: maximize total ~~calorie~~ qty
minimize total calorie

Greedy Sol^n :

1. Create an empty bag
2. while $(x > 0)$
   {
   find the lagest coin c
   at most x ;
   put c in the bag ;
   set x = x − c ;
   }
   } Return coins in the bag

(after end of loop, bag will contain fewest no. of coins)

Eg: say X = 8, optimal sol^n ~~$5~~ ~~$2~~ ~~$1~~

But greedy soln may not give correct sol^n everytime

Eg: $1, $4, $5, $10

greedy gives → 5, 1, 1, 1

But actual ans should have been: 4, 4

NOTE: Brute force always gives 100% correct sol^n/
Greedy → good soln → but may not give correct result all times

Scanned by CamScanner

→ We'll apply greedy only if we don't have any better options known.

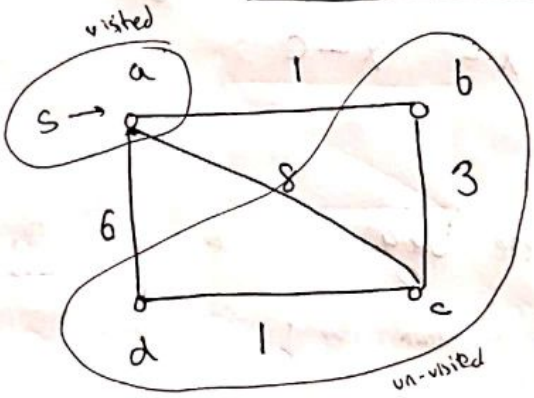
But there are some algo for which greedy is always correct

2 algo to be discussed in syll :

① Dijkstra's shortest path algo
(if graph is connected & undirected & all edges are non-(-ve))

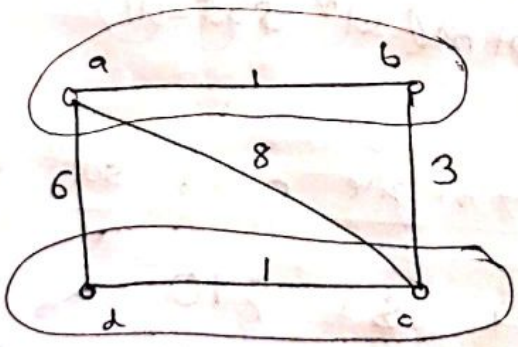
② Prim's MST finding

DIJKSTRA'S SHORTEST PATH ALGO :

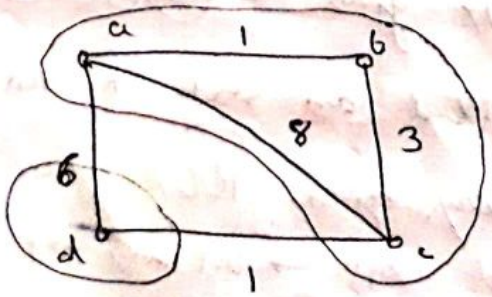


You have to cross the river through these bridges. which bridge ya will choose?

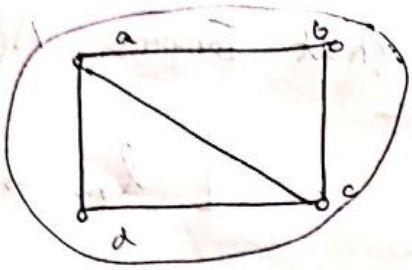
'ab' here
So now a & b will belong to 1 partition



① Everytime you are maintaining the order of visited vertices



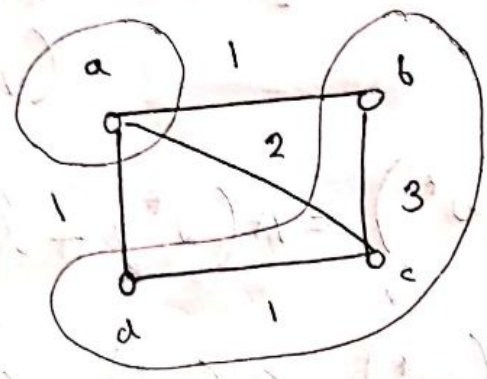
Scanned by CamScanner

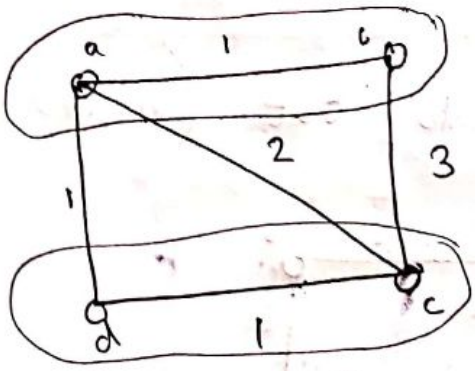


$a \to b \to c \to d$

$1 + 3 + 1 = 5$

Eg:





$a \to b$

Now choose a c

CTI (till BFS (today))

Dijkstra's Shortest Path Algo :

22/02/19.

Outline (this does not use any spcl. data str)

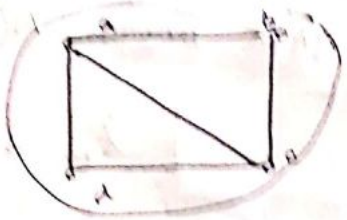
Initialization:

$X = \{s\}$ [vertices processed so far]

$A[s] = 0$ [Computed shortest path distances]

$B[s] =$ empty [Computed shortest path upto s]

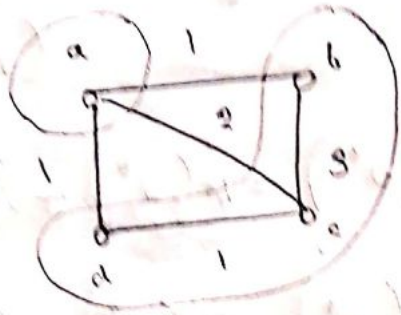
Scanned by CamScanner

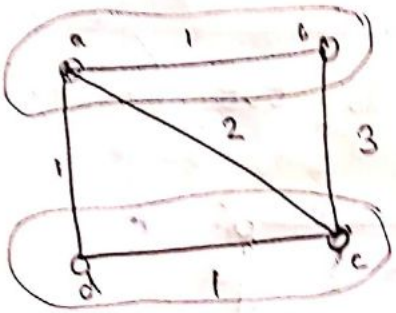


a → b → c → d

1 + 3 + 1 = 5

Eg.





a → b

Now choose a c

CT1 (till BFS (today))

Dijkstra's Shortest Path Algo

22/02/19

Outline (this does not use any spcl. data str)

Initialization:

$X = \{s\}$ [vertices processed so far]

$A[s] = 0$ [Computed shortest path distances]

$B[s]$ = empty path [Computed shortest path upto s]

while $|X| \neq |V|$ (whole loop in total runs $O(n)$ times)

① — among all edges $(v,w) \in E$ with $v \in X$, $w \notin X$

Scanned by CamScanner

pick the w_i that minimizes $A(v) + l_{v,w}$

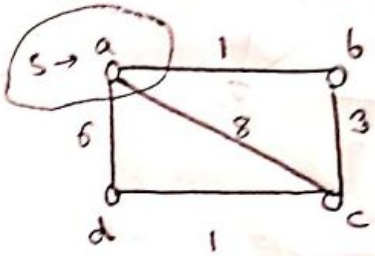
[$l_{v,w}$ → weight of edge (v,w)]

— add w^* to X [if the selected edge (v^*, w^*)]

— set $A(w^*) = A(v^*) + l_{v^*w^*}$

— set $B(w^*) = B(v^*) + l_{v^*w^*}$

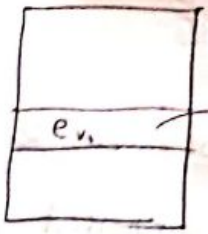
We are dividing the graph into cut (visited & non-visited)



& go on extending the visited set

Using Adj. List:

Say, $X = \{s\}$ → u need to scan the row in the adjacency list to know how many edges are there (hv to find min^m amongst them)



→ no. of entries in row corresponding to s = no. of edges attached to it

So for line ① $X = \{s\}$ takes $O(e_{v_1}) + O(e_{v_1}) = O(e_{v_1})$

(to scan all edges; to find min^m)

→ $X = \{v_1\}$ $O(e_{v_1})$

→ $X = \{v_1, v_2\}$ $O(e_{v_1} + e_{v_2})$

Scanned by CamScanner

→ $|x| = n-1$

$O(e_{v_1} + e_{v_2} + \dots + e_{v_{n-1}})$

→ while loop runs $O(n)$ times

∴ Total running time $= O(n) + O(e_{v_1}) + O(e_{v_1} + e_{v_2}) + \dots + O(e_{v_1} + \dots + e_{v_{n-1}})$

$= O(m)$

∵ $\sum d_i = O(m)$

∴ Total running time $\leq O(n) + O(m) + O(m) + \dots + O(m)$

$\leq O(n) + O(mn)$

$= O(mn)$

Problem is: if it is a densely connected graph then $m = \frac{n(n-1)}{2} \Rightarrow O(n^3)$

If loosely connected graph, $m = n-1$

$\Rightarrow O(n^2)$

So we'll try to improve this, with same concept but using:

① Linear Queue (slight improvement)

② Priority Queue (large improvement)

Using Queue

(also computes: shortest dist. from source to all possible dest^{n}s)

Graph having no. of vertices & no. of edges

```
Dijkstra (G, W, S)
{
    for (each u ∈ V)                    O(n)
    {   d(u) = ∞                        (initially all nodes are at infinity)
        color[u] = white
    }
```

Scanned by CamScanner

for linear → O(c)

```
d[s] = 0
pred[s] = NIL        → storing prev. node
Q = (queue with vertices)   [either linear or priority
while (non-empty (Q))
{
    u = Extract_min (Q)        → for linear O(n²)
    for (each v ∈ adj(u))
        if (d[u] + w(u,v)) < d[
        {
            d[v] = d[u] + w(u,v)
            decrease_key (Q, v, d[v])
            pred[v] = u
        }
    color[u] = black
}
```

(*) for priority queue: binary heap (O(n log n))

evaluating this portion, irrespective of the while-loop

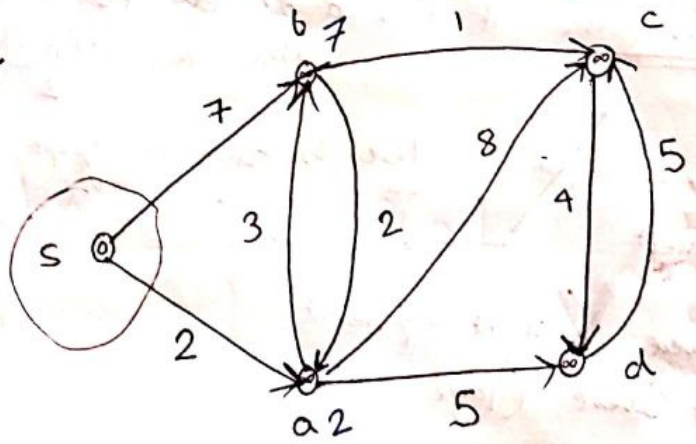
This will be executed O(m) times.

So, O(m) × (comparison + assignment + insertion into Q)

for linear:	for pq:
O(m)	O(m) × O(log n) = O(m log n)

for pq: ... O(log n)

Eg:



This is a greedy decision based. Something is visible to ... Something not visible ...

v	s	a	b	c	d
d[v]	0	2	7	∞	∞
pred[v]	nil	s	s	nil	nil
color[v]	B	W	W	W	W

Queue:

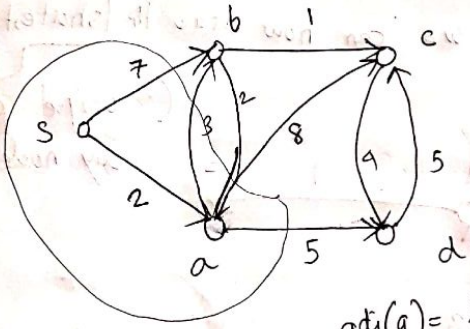
v	a	b	c	d
d[v]	2	7	∞	∞

(lets assume linear queue)

a should be chosen now & removed from queue

Scanned by CamScanner

Now,



adj(a) = {b, c, d}

v	s	a	b	c	d
d[v]	0	2	5	10	7
pred[v]	nil	s	a	a	a
Color[v]	B	B	W	W	W

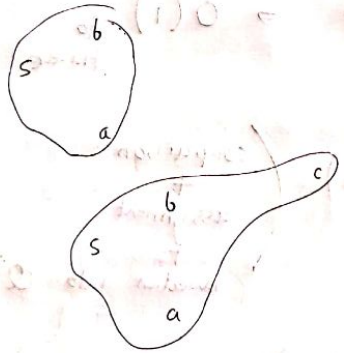
Now, b should be chosen

queue:

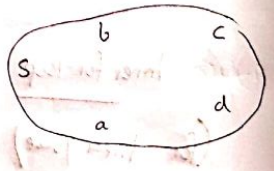
v	b	c	d
d[v]	~~7~~ 5	~~∞~~ 10 (2+8)	~~∞~~ 7

(curr wt, wt of edge)

Then



⇒



Finally

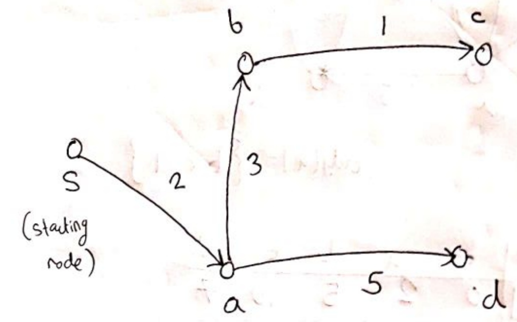
v	s	a	b	c	d
d[v]	0	2	5	6	7
pred[v]	nil	~~s~~ s	a	b	a
Color[v]	B	B	B	B	B

queue : Q = 0

Scanned by CamScanner

Using the pred[v], we can now draw the shortest [illegible] tree

(∵ unique path [illegible] any node [illegible])



hash with pred[v] to get next vertex

Analysis : (from [illegible] of algo)

By analysis, if nothing mentioned, we mean worst case analysis

Initially Q is empty ⇒ $O(1)$ to traverse (★ (algo))

Inner for-loop: (for linear queue)

$O(m) \times$ (comparison + assignment + insertion into Q)

insertion into Q → for linear queue $O(1)$

$= O(m)$

∴ Total running time for linear queue

$= O(m) + O(n^2)$ → Extract-min (a)

$= O(n^2 + m)$

Scanned by CamScanner

Using Priority Queue :

(*) requires building heap now

So $O(n \log n)$

∴ Total running time

$$\leq O(n) + O(n \log n) + O(n) + O(m \log n)$$

$$\leq O(n) + O(n \log n) + O(m \log n)$$

$$= O((m+n) \log n)$$

If graph is weakly connected, $m = n-1$

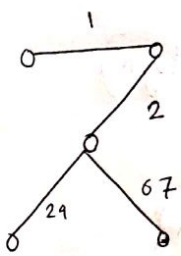
$$\Downarrow$$

$$O(n \log n)$$

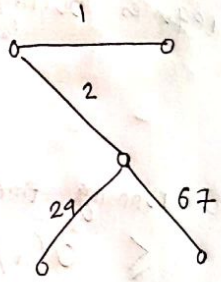
If graph is strongly connected: $O(n^2 \log n)$

Scanned by CamScanner

Spanning trees



total weight
= 94



total weight
= 94

Brute-force approach: Generate all possible spanning trees & find the one with the min^m weight

~~while (not empty~~

```
Prim (G, w, s)
{
    for (each u ∈ V)
    {
        d(u) = ∞
        color [u] = white
    }
    x = {s}
    d[s] = 0
    pred[s] = NIL
    Q = (queue with vertices)
    while (non-empty (Q))
    {
        u = extract_min (Q)
        for (each v ∈ adj(u))
```

Scanned by CamScanner

we don't take length from source

```
if (w(u,v)) < d[v]) & v ∈ V-X    (v is in heap)
{
    d[v] = w(u,v)
    insert (Q, v, d[v])
    pred[v] = u
}
color[u] = black
X = {x ∪ u}
}
}
```

The time complexity of this algo will be similar to the prev version

2 types of analysis : ① using linear queue
② using priority queue

Scanned by CamScanner