* Develop an algorithm or pseudocode to sort an array with a time complexity of O(nlogn)?

One of the most efficient algorithms with a time complexity of O(nlogn) for sorting an array is merge sort.

Algorithm:

(1) Merge_Sort (arr, beg, end)
(2) low = first element, beg
high = last element, last
(3) if (low < high)
{ (4) mid = (high + low)/2 (take floor value)
(5) (repeat) merge_sort (arr, beg, end)
(6) Merge_Sort (arr, mid+1, end) }
(7) Merge(arr, first, last, mid)

Time Complexity of (quick sort)

(a) Best case:

(I) Pivot at the mean position in the list

(II) Partition into 2 balanced parts (n divid
by 2) এবং সময় 2 বার বের হয়।

(III) Partitioning operation visits each element in the array once.

(IV) $O(n \log_2 n)$

Worst case:

Array already sorted, (array 2 বার কোন হলে

(I) Pivot is the smallest / largest element in the

(2) Partition into 1 and $n-1$ elements.

(3) $O(n^2)$

Average Case: (মাঝে মাঝে 2 ভাগে ভাগ হবে

(i) Select pivot uniformly at random Position.

(ii) $O(n\log_2 n)$

* Both merge sort and quick sort has same time complexity but merge sort also required space complexity as it's need another array for sorting. But quick sort doesn't need it. Space complexity of quick sorts is $O(n)$

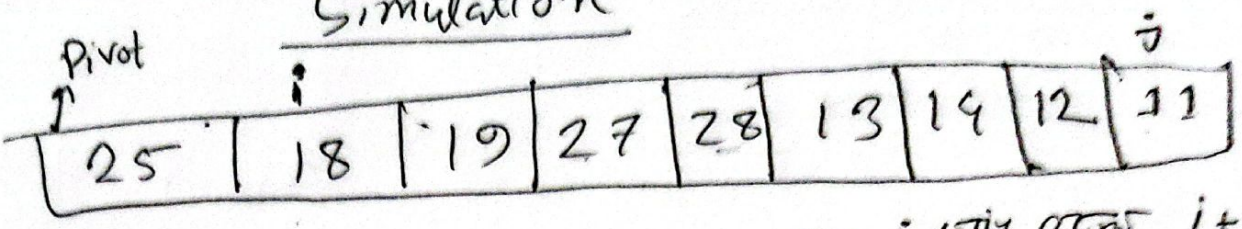
$\log_2 n$)

pivot =

Here $i = pivot + 1$; J = last index

i = lof
j = right

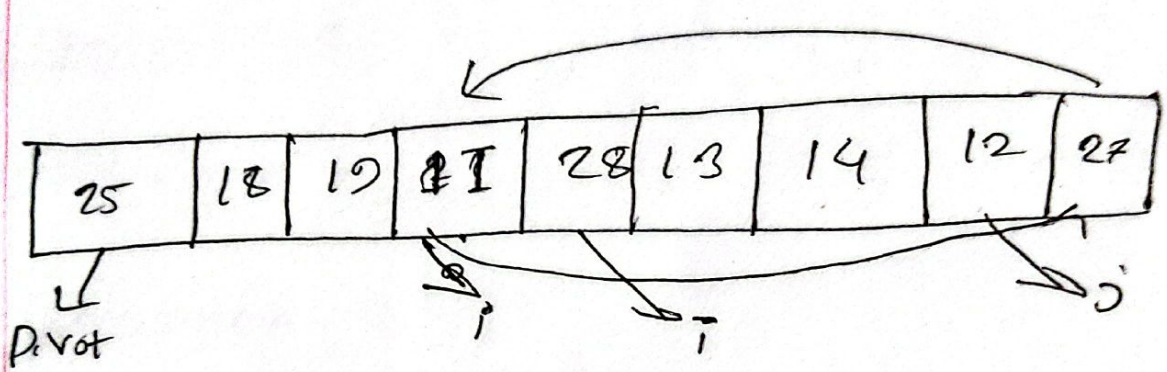
Simulation

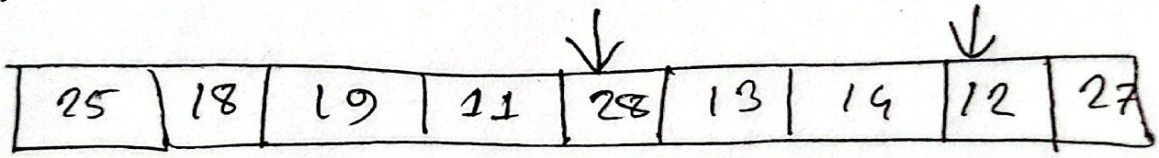


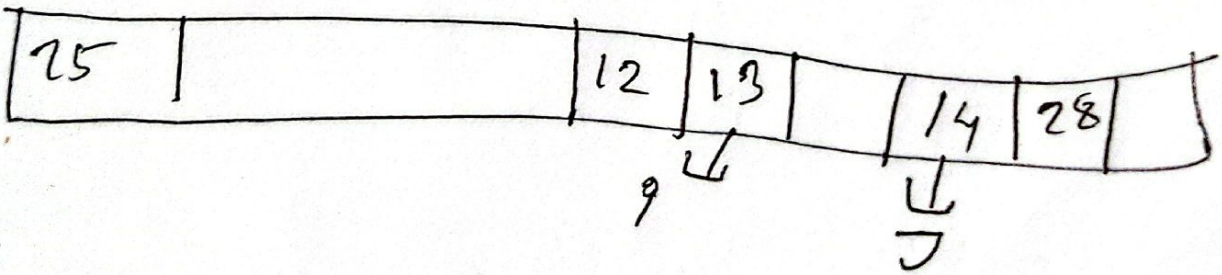
① if $i <$ pivot i++ (pivot এর থেকে i যতো বড় হবে i++

② if $i >$ pivot && $J <$ pivot pivot থেকে i বড় এবং pivot থেকে J ছোট হলে
Swap and i++ , j++ i, J swap and i++, J

③







Dynamic Programming

* 2টা sequence দেয়া আছে এর যদি এদের Lcs বের করতে বলে তাহলে Lcs এ length কখনোই এর 2 sequence এর কোনো একটার exceed করবেনা।

Difference between Subsequence and Substring

Subsequence	Substring

x_i = String 1
y_j = " 2

	y_j	A	C	G	T	C	C	G	C	T	C	G
x_i	0	0	0	0	0	0	0	0	0	0	0	0
A	0	↖1	←1	←1	←1	←1	←1	←1	←1	←1	←1	←1
T	0	↑1	↑1	↑1	↖2	←2	←2	←2	←2	↖2	←2	←2
C	0	↑1	↖2	←2	←2	↖3	↖3	←3	↖3	←3	↖3	←3
T	0	↑1	↑2	←2	↖3	←3	←3	←3	←3	↖4	←4	←4
C	0	↑1	↖2	←2	↑3	↖4	↖4	←4	↖4	←4	↖5	←5
T	0	↑1	↑2	←2	↖3	↑4	←4	←4	←4	↖5	←5	(←5)

* The last element from the table will be the length of LCS

* row এবং column কে check করতে করতে যাবে যদি match পাই তাহলে ঐ match cell এর upper left cell এর সাথে 1 যোগ করবে

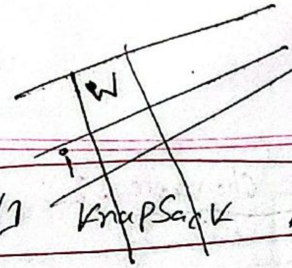
To find those characters:

(i) Follow the arrow

(ii) when you found the diagonal arrow check both row and column characters if they are same keep that character

(iii) repeat i and ii

→ TCTTCA



0/1 Knapsack Algorithm

1) Define DP Table:
   (i) n be number of items
   (ii) w be maximum capacity of ks
   (iii) create a 2d dp where dp[i][w] represents max value can be achived with the first i item and a knapsack capacity of w.

2) initialize DP table:
   - for i=0 and w=0
     Initialize dp[0][w] = 0 for w
     dp[i][0] = 0 for i

3) Fill the DP table:
   (i) iterate from i from 1 to n
   (ii) iterate through each capacity w 1 to W
   (iii) if items weight weight[i-1] is greater than w
   don't include item dp[i-w] = dp[i-1][w]
   (iv) otherwise
      (i) not include if dp[i][w] = dp[i-1][w]
      (ii) Include if dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i-1]] + value[i-1])

0/1 Knapsack

→ Dynamic Programming

প্রতি cell এর মধ্যে Profit বসবে

Item/ob numbers :	0	1	2	3	4	5
weight :	0	3	1	2	4	6
Value / Profit :	0	7	2	1	6	12

7+2 = 9
[9 7 7]

→ W

i ↓	weight 0	1	2	3	4	5	6	7	8	9	1
items number (i) 0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	7	7	7	7	7	7	7	
2	0	2	2	7	9	9	9	9	9	9	
3	0	2	2	7	9	9	10	10	10	10	1
4	0	2	2	7	9	9	10	13	15	15	
5	0	2	2	7	9	9	12	14	15	19	
6	0										

7>0

যেই weight এ যেই প্রফিট থাকবে সেই prev row এর 1st element এর সাথে যোগ হবে [0+7] = 7

Here [15 > 14] এজন্য cell এ 15 বসবে

table

① Each এর জন্য weight & Object name রাখবো.

11)

1) Explain the Similarities and differences between D&C and Dynamic Programming.

Divide and Conquer:

Approach: D&C involves breaking down a problem into smaller and more managable subproblems and solving this sub problem recursively

Divide: The divided problem is independent Subproblem and has the same type as origi

Conquer: Solve this subproblems, recursively, as sub Problems are small enough it can be directly computed,

Combine: If we combine together the solution of Subproblem we'll get the original problem sol

Dynamic Programing:

Approach: Dynamic Programming solves problems by breaking them into smaller simpler subproblem, but different from Dsc it solves each sub Problems once and store them in array. So it doesn't need to recompute.

Overlapping: DP usually deals with problem that ~~needs~~ exibit overlapping. Same sub problem occurs multiple time so it solves each of those reapeating subproblem, and re-use.

Memoization: DP can use either memoization ↑ UP-down (top-down) or tabulation (bottom-up) approach.

Memoization involves storing the results of each sub problem in memory. and if ~~store~~ checks before computing if the solution already exist before we computing.

Optimal Substructure;

Differences

	DP	Dsc
(I)	Deals with overlapping sub problem	(I) Doesn't deals
(II)	Dsc need recomputation	(II) Stores solves using mem So do if the solution already ex doesn't need reco
(III)	require storage to store solution of sub prob	(III) not needed,
(IV)	tends to break Problem into smaller size	(IV) focuses on reusing solution of sub problem DP

Similarities

DP	

involves.

(I) Both Paradigms ~~Problem~~ breaking down Problem into smaller, more managable subproblems

(II) Both paradigms often recursion to solve subproblem.

(III) optimal solution to the original problem depends on the optimal solution to its subproblems

Pseudo code for quick sort

```
int partition (int a[], int low, int high) {
  int pivot = a[low];
  int i = low + 1;
  int j = high;
  while (i < j) {
    while (a[i] < pivot && i < high) {
      i++; }
    while (a[j] > pivot && j > low) {
      j--; }
    if (i < j) {
      swap (a[i], a[j]);
      j--;
      i++; }
  }
  swap (a[low], a[j]); return j;
```

→ Greedy

Knapsack Fractional

n = 7
max weight = 15

Objects	1	2	3	4	5	6	7
Profit	12	5	16	7	9	11	6
weight	3	1	4	2	9	4	3
Profit per wt (P/W)	4	5	4	3.5	1	2.75	2

Objects	Profit	weight	Remaining
2	5	1	15 - 1 = 14
1	12	3	14 - 3 = 11
3	16	4	11 - 4 = 7
4	7	2	7 - 2 = 5
6	11	4	5 - 4 = 1
7	6 × 1/3 = 2	1	1 - 1 = 0

Profit × 1/taken weight

total Profit = 56

* Greedy about weight
* " " Profit
* Greedy about Both

Pseudo Code:

```
double GreedyKS (double c, int T[n][3], int i, int n)
if (c<=0 || i>n)
    return 0;
if (c < T[i][2]) // capacity less than current i
{   double P = c / T[i][2] * T[i][1]
    return P;
}
return T[i][1] + GreedyKS (c - T[i][2], i+1, n);
}
int main() {
int T[3][4] = { // 0  1  2  3
                // id, v, w, v/w
                {1, 60, 10  6
                {
```

```
double C = 50 // bag capacity
double Profit = Greedyks(C, T, 0, 3)
cout << Profit << endl;
```

Start idx, last Colm

Algorithm:

(I) Calculate the value to weight ratio for each item

(II) Sort the item in decreasing order of their value-to weight ratio.

(III) Initialize the total item in Knapsack 0 and create an empty KS.

(IV) Iterate through sorted item.

(1) if the weight of current item is less than remaining capacity, at that item into KS and subtract its weight from remaining weight

(2) Otherwise add a fraction item to the KS where the fraction is equal to the remaining capacity.

(V) The algorithm stops when all items have been Processed or the remaining capacity becomes

Coin change Problem

Algorithm For finding minimum number of coin

1. Select coin/element with largest value that is less or equal to amount (max value ≤ amount)
2. If we can't take the largest available coin anymore, move to the smaller (second) largest element and so on
3. Reapeat steps (1 and 2) until the change amount is complete, (recursion)

Q: Assume we have an unlimited number of coins of various denominations.

Objectives Payout a given sum S with the smallest number of coins possible.

The greedy Coin changing algo:

(I) While S > 0 do {

(II) Choose a coin of highest denomination but less than or equal to S. Until we can't take it anym

(III) Subtract the chosen coin from S

(IV) Reapeat }

Example 3:

Given notes = 1, 5, 6, 9

have to find minimum number of coin/notes to make 11.

As the highest possible available coin is 9 so we take 9 but then we need 2 takes 1 and 1

So input = 11

Output = 3 (9, 1, 1) but it's not optimal

We can make 11 with 5 and 6. so optimal solution

So sometimes greedy algo will not give optimal solution.

Coin change Problem

Algorithms For finding minimum number of coin

1. Select coin/element with largest value that is less or equal to amount (max coin value ≤ amount) c_i
2. If we can't take the largest available coin anymore, else move to the smaller (second) next coin largest element and so on.
3. Reapeat steps (1 and 2) until the change amount is complete. (recursion)

Assume we have an unlimited number of coins of various denominations.

Objective: Payout a given sum S with the smallest number of coins possible.

The greedy Coin Changing algo:

Total amount (S)

(I) While S > 0 do {

(II) Choose a coin of highest denomination but less than or equal to S. Until we can't take it anymore

(III) Subtract the chosen coin from S.

(IV) Reapeat.

Example 3:

Given notes = 1, 5, 6, 9 of coin/notes to have to find minimum number to make 11.

11 − 9 = 2, 2 − 1 = 1, 1 − 1 = 0

As the highest possible available coin is 9 so we take 9 but then we need 2 take so we take 1 and 1

So input = 11
Output = 3 (9, 1, 1) but it's not optimal as we can make 11 with 5 and 6. So optimal solution will not give optimal solution.

So sometimes greedy algo will not give optimal solution.

Pseudo code:

```
Const int numCoins = 5;
int output [numCoins]; // to store number of coins
int greedy coin change (C[], int n, int i)
{ if (n==0) { return 0; }
if (i < numcoins && C[i] <= n)
{ output [i] ++;
return 1 + greedy coinchange (C, n-C[i], i);
} else
return greedy coinchange (C, n, i+1);

int main {
int numcoins = 5
int C[numcoins] = {50, 25, 10, 5, 1};
n = 87 ;
for (int i=0 ; i < numcoins ; i++) { output[i] = 0;)
result = greedy coin change (C, n, 0);
cout << "Minimum number of coin " << result << endl;
cout << " for coin used " << endl;
for (int i=0 ; i < numcoins ; i++)
{ if (output [i] > 0)
    cout << C[i] << "x" << output [i] <<
```

Job sequence with Deadline

(1) Sort Profit into descending/decreasing order & jobs must

(2) Initialize an empty schedule where we'll assign the jobs.

(3) Iterate through the list of sorted jobs.

a) Assign each job to the last available time slot that satisfies the deadlines.

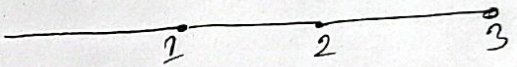
b) if the slot is already occupied move to the previous slot until a suitable slot is found

c) if no slot is found ignore the job

1) Sort all the jobs Profit in descending order.

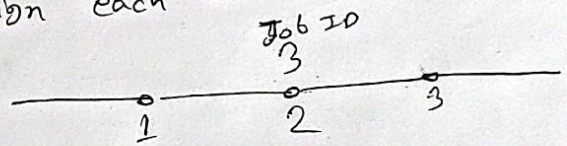
JID	3	1	2	4	5	
Deadline	2	2	1	3	1	
Profit	100	90	80	50	40	

2) Initialize a schedule where we assign the job

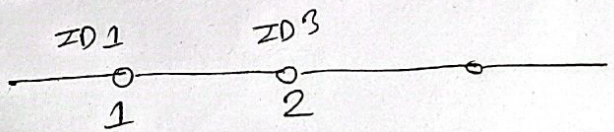


Here highest deadline is 3

3) (a) Assign each job to last available time slot



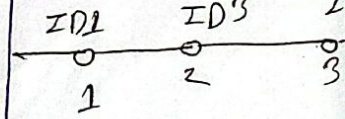
(b) If slot already occupied move to previous slot



(c) If no slot found ignore the job

So Job 2 is ignored

4) For ID 4 Deadline is 3 and 3 is available



Total Profit : The total Profit is the sum of the Profits of the assigned jobs.

$\therefore$ Total Profit $= 100 + 90 + 50$

$= 240$

How to decide which choice is optimal?

(i) Greedy made a sequence of choices At each step it makes choice that looks best at the moment. Choices are made locally with a hope that this choices will lead to a globally optimal solution. It choices made at each step depends on current state without regarding future consequences

(ii) DP is used when Problem can be broken down into overlapping sub problems. it solves problems by breaking down them into smaller simpler sub problem and each subproblem and store the result by using top-down and bottom up approach so if the same problem arise again it skip executing recalculation and directly place the answer. DP helps to reduce during calculating redundancy. Dp ensures to give optimal solution if the problem has a solution and shows optimality

Date: Page 23, 22

What are the key differences between the dynamic programming algorithm and the greedy approach? Why DP is better than Greedy algorithm?

Both DP and greedy are strategies to solve Optimization Problem. But DP is used when Problem can be broken down into overlapping Subproblems. it solves problem by breaking down them into smaller simpler sub problem and Solves each subproblem then constructs it

Greedy Algorithm: this algorithm makes locally Optimal choices at each step with the hope that this choices will lead to a globally optimal solu The choice made at each step depends only o the current state without regard to the Consequences. UNLIKE DP they do not consid previous choices once theyre made

Dynamic Programming Guarantees finding Optimal Solution if the Problem exhibits the Principle of optimality and can be broken down into overlapping subproblems.

But Greedy algorithms do not always guarantee an optimal solution. They might find a locally optimal solution that doesn't necessarily lead to the best global solution.

Dp involves solving subproblems and storing their solutions to avoid redundant computations. It typically uses bottom up or top-down to convert larger problem into smaller.

Typically DP have higher time complexity due to need to solve all the subproblem, but time complexity can be reduced using memoization or tabulation.

Dp is better over Greedy when,

(i) Problem can be divided into overlapping sub-problems. and exibits Principle of optimality.

(ii) Problem requires optimal solution.

(iii) When Problem size is small and time complexity is manageable.

Dynamic Programming is generally Preferred Over Greedy when optimality is required and Problem can be broken down into overlapping sub problem.

Greedy is chosen when simplicity and efficiency are more critical than finding optimal Solution.

Why Greedy better than Dynamic Programming

Simplicity: Greedy algorithms are often simpler to implement and understant compared to dynamic programming. which involves breaking the problem into sub then solving iteratively. if a problem can be efficiencty solved with straight forward Greedy it might be Preferable due to its simplicity.

Greedy have lower time complexity as it makes decisions based on current state without considering future. if the problem is large. Greedy is satisfactory, due to its time efficiency

If finding optimal solution is not necessary Greedy might suffice. Greedy often Provide Suboptimal solution, it can be acceptable depending on problem objectives

it has lower space complexity than DP as it doesn't require memoization.

Greedy is suitable for those Problem Where making locally optimal choices leads globally Optimal Solution. if Problem exhibits Greedy Choice Property Greedy approach is best.

But its essential to recognize Greedy it only not always guarantee optimal solution but On the other hand DP guarantees optimal Solution.

When the problem requires simplicity, efficiency and acceptable solutions (suboptimal) over optimal Greedy is Preferable.

Why does Dynamic Programming is called "clever brute force", Describe an example and reasons of using DP

It often reffered as clever brute force because it Optimizes the Proces of solving a problem by Combining the efficency of DP with the exaustive Search approach of bruteforce. The classic exam of DP is Solving Fibonacci efficiently. While a nai recursive approach to calculate fibnacci numbers exponential time complexity. DP reduces the time complexity to linear by storing previously computed values. it cleverly avoids redundant comput by storing solutions to subproblems!

Combining the bruteforce approach "considering all possibilites" with clever reasons for DP by avoiding redundant computations through clever optimization strategy DP is called clever bruteforce.

Discuss time complexity in Algo. Explain why Time Complexity measure important.

i) Performance Evaluation: By Analyzing TC developer can select most efficent algo for particular problem. For large DS small difference lead significant d

ii) Scalability: helps to understand how a algo's performance scales with size of input

iii) Resource management:

iv) Optimization:

Shortest Path

There are 3 Shortest Path variants:

① Single Source Shortest Path: from one vertex to every other.

(i) Dijkstra Algo → Non-negative weight (Cost)

(ii) Bellman ford ⇐ No-negative cycle

Example: Dhaka থেকে যেকোনো জায়গার Shortest Path.

② Source-Sink Shortest Path: From one vertex to another.

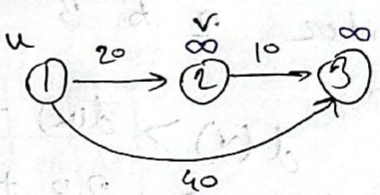
③ All Pairs Shortest Path.

Follows Greedy

Dijkstra Algorithm

Single Source Shortest Path

Relaxation:



① If I want to go to 2

then, let $d(v) > d(u) + c(u,v)$

$\infty \quad 0 \quad 20$

Then, $d(v) = d(u) + c(u,v)$

$= 0 + 20$

$\therefore d(v) = 20$

(ii) To go 1 to 3

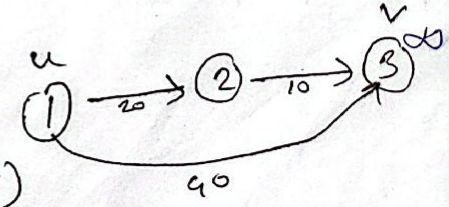
$d(v) > d(u) + c(u,v)$

$\infty \quad 0 \quad 40$

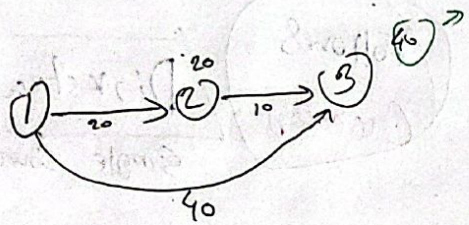
$\therefore d(v) = 0 + 40$

$= 40$

as i am already standing on d(u) so cost/wt of d(u) is 0



(III) Now if i want to go 1 to 3 through 2



Relaxation :

(I) Cosider 2 to 3 as 2 is in middle of 1,3

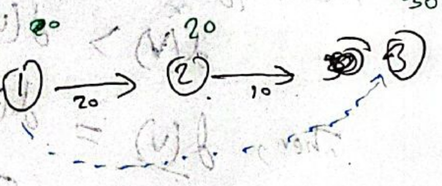
$d(v) > d(u) + d(u,v)$

$\therefore$ $d(v) > d(u) + d(u,v)$

$= 40 > 20 + 10$

$\therefore 40 > 30$

So $d(v) = 30$



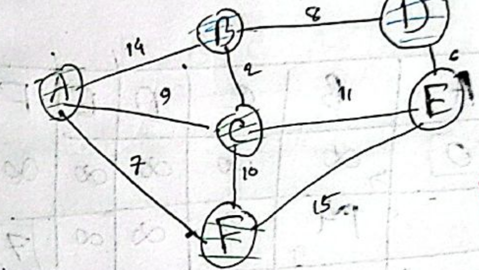
ede cesoR current t path

$\leftarrow \pi[V] = 2$

Relaxation $\longrightarrow$ if, $d(v) > d(u) + c(u,v)$

then $d(v) = d(u) + c(u,v)$

For un Directed :



Here, $d(v) = d(u) + d(u,v)$

$= 9 + 10$

$= 19 > 9$

Source

	Visited vertex	A	B	C	D	E	F
A		[0]	∞	∞	∞	∞	∞
F	A		14	9	∞	∞	[7]
C	F		14	[9]	∞	22	
B	C		[11]			20	
D	B				[8+11]	20	
E	D					[20]	

$7 < \infty$

$9 < \infty$

Here $14 < \infty$

→ সব থেকে ছোট, F = 7

C = 9

$= 9 + 2 = 11 < 14$ for B

again,

$9 + 11$

$= 20$ থেকে $22 > 20$

→ D = 19

$= d_u + c(u,v)$

$= 19 + 6$

$= 25 > 20$

(I) A to D Shortest Path বের করব

Visited	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
F		14	9	∞	∞	7
C		14	9	∞	22	
B		11	∞	∞	20	
D			12	20		
E				12 20		

(i) First take D

(ii) Backward যেয়ে Previous Row তে change আছে এজন্য এই Row তে যেটি select হয়েছে তাই নিবে

D B

(iii) আবারো B change এখানে C selected,

D B C

(iv) No change.

(v) Change. A

D B C A

(vi) যদি Previous row তে কোনো change না থাকে তাহলে Continue Backtrack

Back Tracking

N-Queens Problem ; Time Complexity: $O(n!)$

* first queen in 1st row
  2nd " " 2nd "
  3rd " " 3rd "
  4th " " 4th "
* To check queens locations we have to check their Column.
* Avoid 2 queens in same row/column

Condition

* Column ; $C_1 == C_2$ (not safe)
* Row ; $R_1 == R_2$ (not safe)
* Diagonal ; $abs(C_1 - C_2) == abs(R_1 - R_2)$

N queens

Algorithm

1) **Initialize:** Start with an empty N×N Chess Board. initially set all cell to ~~infinity~~ 0

2) **Place the queen:** Place the first queen in the first Row.

3) **Check for conflicts:** Check if the Placement of queen is valid. (it doesn't attack other queens)
   - no 2 queens can be in same row,
   - no 2 " " " " " column.
   - no 2 " " " " " in " Diagonal.

Recursive call: If current Placement is valid move on to next row and repeat 2-3. If current Placement is not valid backtrack to the previous row, and try a different column (move to forward column)

6) **Solution found:** If a queen has been Successfully Placed in each colum a solution has been found and can be reported.

7) Report all the solutions

Conditions:
- Column: $c_1 == c_2$ (not safe)
- row: $R_1 == R_2$ (not safe)
- Diagonal: $abs(c_1 - c_2) == abs(r_1 - r_2)$

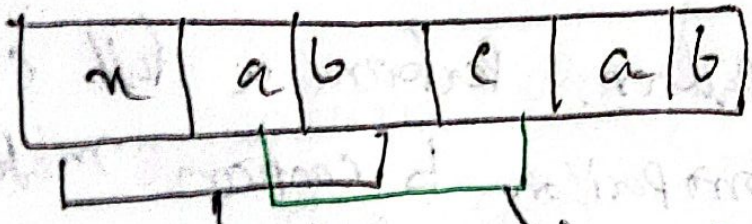
Rabin Karp

Rolling Hash: R/K algo uses rolling hash. Which meant that the algo slides its window over the text. it updates it's hash value incrementally.

- taking into account the removal of the first character and the addition of next character. move the slide right side by excluding first character and including ~~first character~~

Consider :- a=1, b=2, c=3, x= 27

Search (a, b, c) → (a, b c) 3 character
= 1+2+3 = 6



(24+3+2) = 27
27 ≠ 6

(1+2+3)
= 6

Here, Hashes are match, it's a valid match

if a window hash is 6 then start checking individual character.

Algorithm

1) Take a window size of M. (same as finding Pattern)

2) Compute the hash value of Key text

3) if the hash of window and the Pattern is same.
   - compare each character of the window and the Pattern. Perform a full character by character comparison to confirm match.

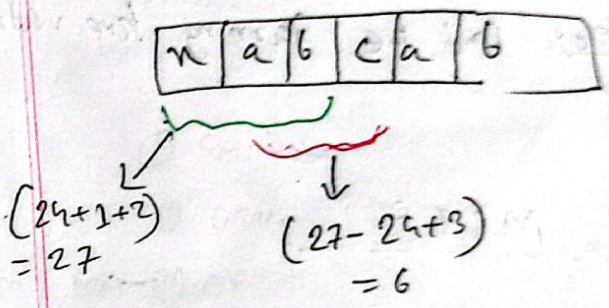
4) if a match found record the starting index of match.

5) if the hash of window and pattern aren't same continue sliding the window one character right

6) Continue until end of the text is reached.

improvements

- Use the hash value to current window calculate the hash value of next window.
- will reduce the time complexity $O(m)$ to $O(1)$



Sliding window operation take $= O(n-m+1)$

match take $O(m)$

Overall complexity $O(n-m+1) * O(m)$

Minimum Spanning Tree

Spanning Tree: A tree that contains all vertices

minimum Spanning Tree: find the spanning tree with minimum weight

applications of MST:

O/1 / LCS

(I) Create a 2D matrix of size n*w (O(1)) time

(II) filling the table takes O(n*w)

(III) overall time complexity O(n*w)

(IV) Space complexity.

For lcs add extra line = (m+1) x (n+1)

Frac Knap

(I) Sorting time complex → O(nlogn)

(II) Greedy choice → O(n)

coin / frac (III) overall time complex → O(nlogn) + O(n) → O(nlogn)

for coin change if m/n significantly small time com → O(n)

Kruskal	Prim
(I) Sorting E edge = O(E log E) (II) Select E edge → O(E) (III) Check v node O(V) (IV) Overall time O(E log V)	(I) Adjacency mat — O(V²) (II) Adjacency List — O(E log

N-queens

(I) All possible arrangements → n^n

(II) Backtrack to reduce → n!

(III) Time complexity → O(n!)

Graph color

Time complexity $\rightarrow O(m^V)$

m = number of color

V = vertex number.

Rabin Karp

Processing $O(m)$

Worst case $O((n-m+1) \times m) + O(m)$

Average case $O(n)$