29-05-25

→ Activity Selection prblm is an optimizati[on] problem, which deals with the selection o[f] non-optimized problem which needs to b[e] Executed by a Single in a given time frame. Each activity is marked by a start & finish time.

→ It might not be possible to complete all the activities ~~still it~~ Since their timings can be collapsed

→ Two activities, I and They are said to be non-conflicting if SI - Start time of I is greater than or Equal FJ - Finish time of J. Where, SI and SJ denotes the start time of I and J activities and FI, FJ finish the time of I, J activities.

→ This greedy approach can used to find the Solution, Since, I want to maximize the count of activities that can be Executed So this approach will chose an activity with Earliest finish time

Start time (s)	Finish time (f)	Activity name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

Sorted:

f	act name
2	a2
4	a3
6	a4
7	a5
9	a1
9	a6

Catalan number

$n = 5$

Catalan

dp = new long [5 + 1]

dp = 6

dp[0] = dp[1] = 1;

for (int i = 2; i <= n; i++)

dp[i] = 0

for (int j = 0; j < i; j++)

dp[i] += dp[j] *

dp[i - j - 1];

dp[1]

dp[2] += dp[0] * dp[1]

~~dp[2] +=~~

dp[2] = 2

dp[2] = dp[0] × dp[1] +
dp[1] × dp[0] +

dp[3] = dp[0] × dp[1] +
dp[1] × dp[0] +
dp[1] × dp[2] +
dp[2] × dp[1] +
dp[0] × dp[2] +
dp[2] × dp[0] +

dp(2) = 2
dp[3]
⋮
dp[5]

$$\boxed{c(n) = \sum_{n=0} c(i)(n-1-i)}$$

i = 0 to n-1

MinCoins

int [ ] coins = { 1, 2, 5 };

int amount = 11

Max Value - ∞
Min Value - ∞

minCoins (coins, amount)

int [] dp = new int [amount + 1]

=) 11 + 1

=) 12

dp[] = 12

Arrays.fill (dp, Integer.MAX_VALUE);

(12, I)

dp[0] = 0;

```
for (int coin : coins) {
    for (int i = coin, i <= amount ; i++)
    {
        if (dp[i - coin] != Integer.MAX_VALUE)
            dp[i] = Math.min( dp[i], 1+dp[i-coin])
```

i <= 11

0 != 5

1+dp[0]

dp [1] = 1

dp[2] = 2, 1+dp[0]

2, 1

dp [2] = 1

dp[5] = ~~dp~~ (5, 1+0

dp[11] = ~~5 2 -1 dp[11]~~

Subset Sum

→ 1[st] testcase

set1 = {1, 2, 1}

Sum1 = 3

n1 = set1.length (3)

List<Integer> subset1 = new ArrayList<>();

SubsetSum (0, n1 (3), set1 ({1,2,1}), sum1 (3), subset1);

i, n, int[], int targetsum

{

if (targetsum == 0 (3)) {

}

if (i == n) { (0 == 3)

}

printSubsetSum (i+1 (1), n (3), set, targetSum (3),

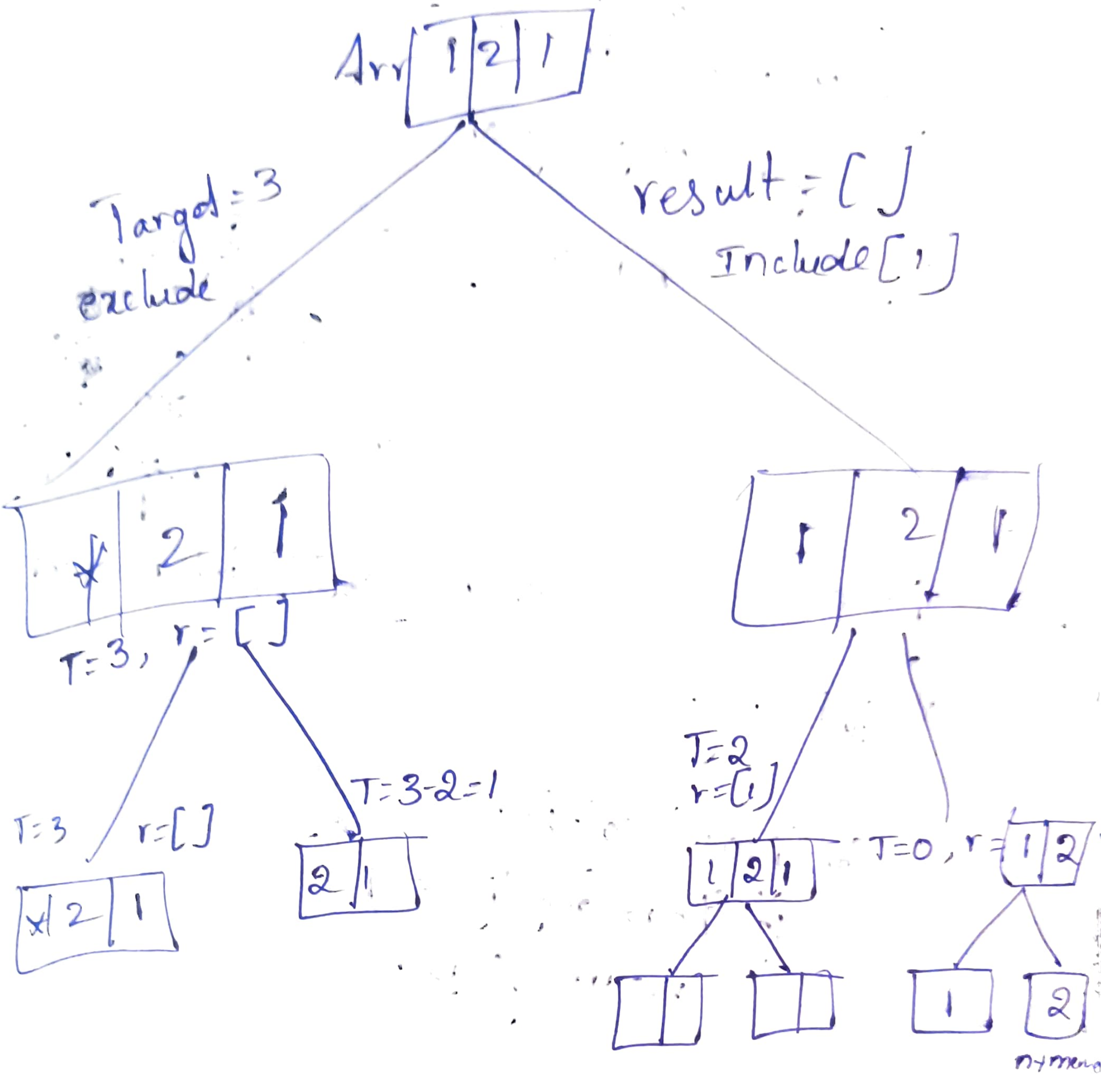
subset);

{

if (set[i] <= targetsum) (1 <= 3)

subset.add (set[i] (1))

State Space diagram

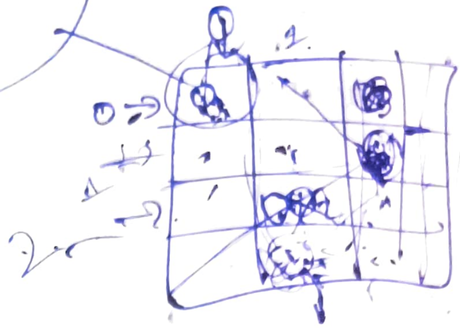


N-Queen

4x4 :

	Q_1		
			Q_2
Q_3			
		Q_4	

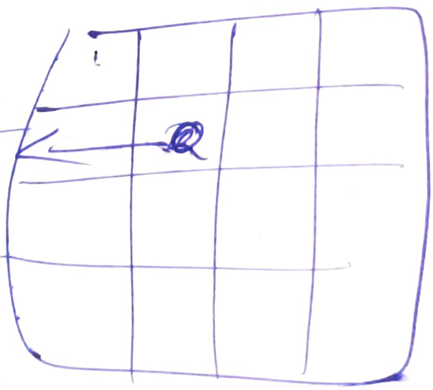
```
for (int i=0; i<N; i++){
    for (int j=0, j<N
        board
```



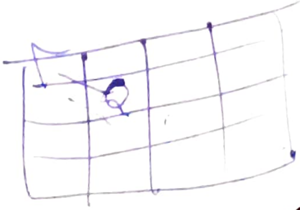
```
boolean isSafe (int board[][], int row, int col)
{ int i, j;

    for (i=0; i<col; i++)
        if (board[row][i] == 1)
            return false;
```

for Ew



```
    for (i=row, j=col; i>=0 && j>=0; i--, j--)
        if (board[i][j] == 1)
            return false;
```



1FC solveNQUtil (board, 0 col)

isSafe (board, 0, 0)

2FC solveNQUtil (board, 1 col)

isSafe (0, 1)

(1, 2) i=1, j=2

3rd FC

ACID properties in DBMS

Atomicity: The Entire transaction takes place at once or doesn't happen at all. Transaction refers to a options of Sequence performed in a single unit

Atomicity means that Either the transaction Completes fully or doesn't Execute at all.

There is no intermediate states

The transaction do not occur properly.

If a transaction has multiple operations and if any of the operation fails the whole transaction is rolled back and doesn't affects the database.

Atomicity avoids partial updates which can lead to inconsistency.

* If the transaction fails ~~after~~ before the completion of T_1. But after the completion of T_2 the database will be in an inconsistent state.
* The Entire process is rollbacked to the original stage.

Before X = 500	Y = 200
Read X = X-100	Read Y
write X	Y = Y+200
X = 400	write Y
	Y = 300

-) Consistency: The database must be consistent before & after the transaction.
-) It Ensures that the database must be in a valid state before & after the transaction.
-) It guarantees that a transaction will take data base from one consistent state to another maintaining the rules & constraints defined for the transactions of data.
-) Isolation:- Multiple transactions occur independently without interference.

This property Ensures that multiple transactions can occur without leading to inconsistent of database & changes that occur in a particular transaction will not be visible to any other transaction until that particular change in the transaction is return to the DB or has been committed.

~~Durability~~

This property Ensures that dirty reads (reading the uncommitted data)

-> non-repeatable trees (data changing b/w the two reads in a transaction

-> Phantom reads: New rows appearing in a result set after the transaction starts

-> Tree wants to transfer 50 from x to y transaction reads the input of y deducts the 50 from X. and add 50 to Y. which makes new X = 450, new Y = 550

-> The X and Y values in the database should be committed to 450 and 550 which is Equal to 1000 and ~~the~~ maintaining the consistency with the start of the transaction.

-> It Ensures that T should not Ensure values X & Y, while the transaction is in the progress. Both the transaction should be independent and be should only seen in final state of the transaction after it commits

-) Durability

The Changes of a Successful transaction occurs Even if the System failure happens Once the transaction has completed the Execution and modifications the updates to the database are shown and return to the this and they persists Even to the System failures. These updates then become permanent and Stored in the non-volatile memory. At this point of failure, the dbms can recover the database to the state it was after the last committed transaction Ensuring no database is lost