DP - 3

Question: Wildcard Pattern Matching

→ Given a string and pattern, check if the string matches the pattern.

→ string → lower case [a - z]

→ Pattern →
a) lower case char [a-z]
b) ? → [Match with any 1 char]
c) * → [Match with 0 or more characters]

Ex1: s = a b
p = a * b ⇒ ✓

Ex2: s = a
p = a * b ✗

Ex3: s = a a b
p = a * b ✓
a

Ex4: s = a a b
p = a ? b ✓
a

s = a x b c

Ex5: s = a a a a b
p = a * b ✓

Ex6: s = a a a a b
p = a ? b ✗

Ex7: s = a b ✗
a ? b

Eg: p = a ? b * c ✓

s = a z

Eg: p = a * * d * * z

s = a xy z c

Eg: p = a ? ? ? c

p_1 = ? p_2 = *

s_1 = a

s_2 = ab

s = abcdef

p = a * d ? f

bc e → abcdef

Case 1: Last char of pattern in alphabet..

s = abcdef

p = a * d ? f

$l_s - 1$ $l_p - 1$

s = abcdex

l_s → string l_p → pattern

if (s[l_s-1] == p[l_p-1])

ans = isMatch (l_s-1, l_p-1)

else

ans = false;

Case 2: Last char of pattern is ?

s = a b c d e f

p = x * c ?

if (p[l_p-1] == '?')

ans = isMatch (l_s-1, l_p-1)

Case 3: Last char of pattern is *

s = a b b a

p = a *

* = b b a

Option1: * consider 0 characters

Option2 * consider 1 or more characters

Option1: * → 0 character.

s = a b b a

p = a *

ans = isMatch(ls, lp-1)

Option2: * → 1 or more characters

ans = isMatch(ls-1, lp);

ans = isMatch(s.length(), p.length())

dp[][]

isMatch(ls, lp) {

if(ls == 0 && lp == 0) return true;

if(lp == 0) return false.

if(ls == 0) {

for(i=0; i<lp; i++) {

if(p[i] != '*') return false;

state of the problem

O(ls * lp) ---

dp[ls][lp] =

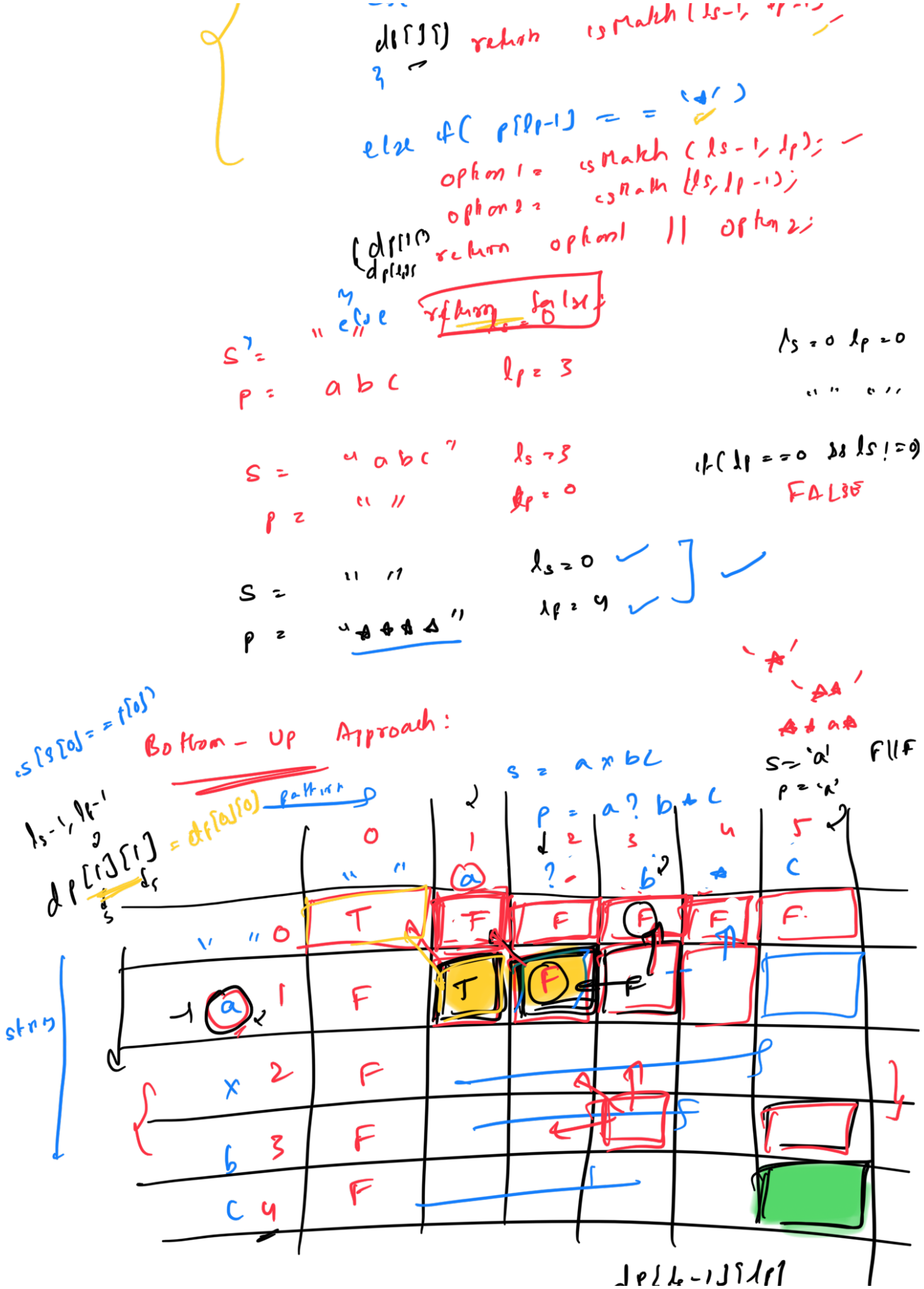
if(dp[ls][lp] != -1) return dp[ls][lp]

if(s[ls-1] == p[lp-1]) {

dp[ls][lp] return isMatch(ls-1, lp-1);

O(1)

} else if(p[lp-1] == '?') {


dp[][] return isMatch(ls-1, lp-1);
else if(p[lp-1] == '*')
opt1 = isMatch(ls-1, lp);
opt2 = isMatch(ls, lp-1);
return opt1 || opt2;
dp[][]
dp[ls][lp]
return false;
s = "" p = "abc" ls = 0 lp = 3
s = "abc" p = "" ls = 3 lp = 0
if(lp == 0 && ls != 0) FALSE
s = "" p = "****" ls = 0 lp = 4
Bottom - Up Approach:
s[s[0] == p[0]
ls-1, lp-1
dp[i][j] = dp[s[0]p[0] pattern
s = axbc
p = a ? b * c
s = 'a' FALSE
p = 'a'
string
T F F F F F
F
F
F
F
dp[ls-1][lp-1]

dp[ls][lp-1]

// fill 1st row & column

for(ls=1; ls<=ls; ls++){

for(lp=1; lp<=lp; lp++){

if(s[ls-1] == p[lp-1])

dp[ls][lp] = dp[ls-1][lp-1];

else if(p[lp-1] == '?')

dp[ls][lp] = dp[ls-1][lp-1];

else if(*)

dp[ls][lp] = dp[ls-1][lp] || dp[ls][lp-1]

}

else dp[ls][lp] = false;

}

O(# states * T.C Per state)

↓

$l_1 \times l_2$ × O(1)

T.C: $O(l_1 \times l_2)$

S.C: $O(l_1 \times l_2)$ → $O(l_1)$

*

$10^3, 10^3$

↓

(1s)

↓

Subsequence

Question: Longest Palindromic Subsequence

S = b e b e e e d

Ans = "eeee" → 4

S = a e d s e a d

Ans = a e d e a → 5

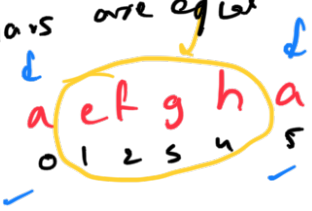
S = a e d e e a d

a e d e a

Ans = 5

Solution 1:

Case 1: Last chars are equal



= 2 + LPS(e f g h)

LPS(i, j) = length of Longest Palindromic Subsequence of substring s[i .... j]

if (s[i] == s[j])

LPS(i, j) = 2 + LPS(i+1, j-1)

Case 2: Last chars are not equal

S =

if (s[i] != s[j]) {

max(LPS(i+1, j), LPS(i, j-1))

lps(i,j) = max(

```
dp[][]
int lps(int i, int j){
    if(i > j)  return 0;
    if(i == j) return 1;
    if(s[i] == s[j])
        return 2 + lps(i+1, j-1);
    else
        return max(lps(i+1, j), lps(i, j-1));
}
```

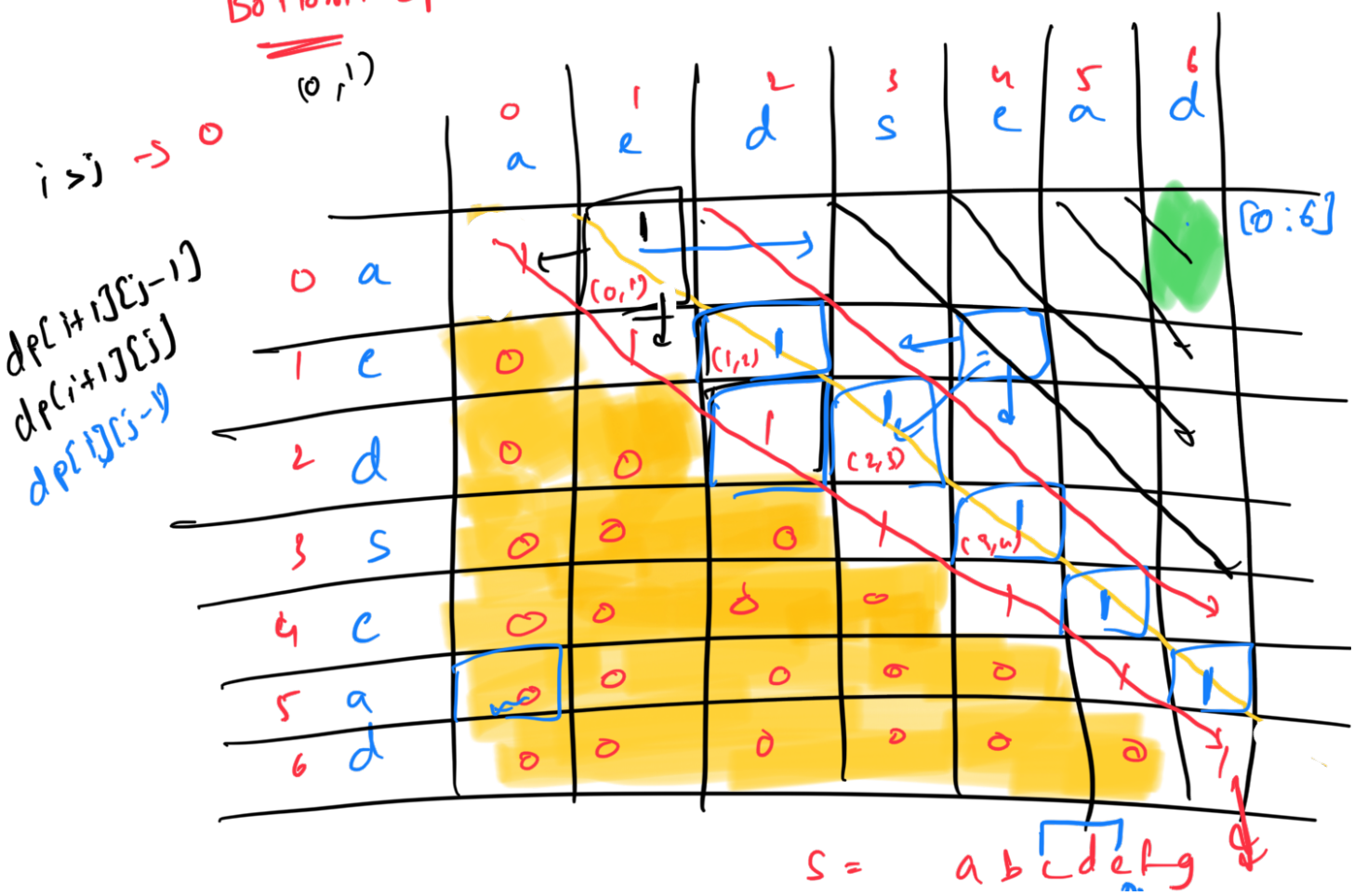
o(1)

T.C: # states × T.C per state

$O(n^2 \times o(1)) \Rightarrow o(n^2)$

L = 1

Bottom-up:



LPS of substring of L = 4

LPS(bcdeb) → 2+2+1 = 5

LPS(cde)

Approach 2:

A = a c d s e a

A' = a e s d e a

} LCS = 2 ("aeda")

LCS(A, A')

Question: Palindrome Partitioning

→ Partition the string such that every substring is a palindrome

s = a b a a b

(a, baab) 1 cut

(aba, a, b) 2 cuts

(a, b, aa, b) 3 cuts

(a, b, a, a, b) 4 cuts

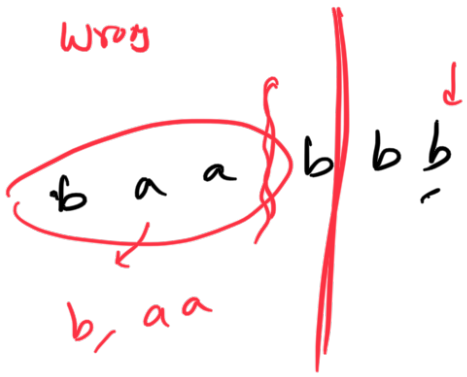
Ex1 a b c d e → 4

Ex2 a b c d e → 0

Ex3 a a a a → 3

Ex4 a b b a c → 1

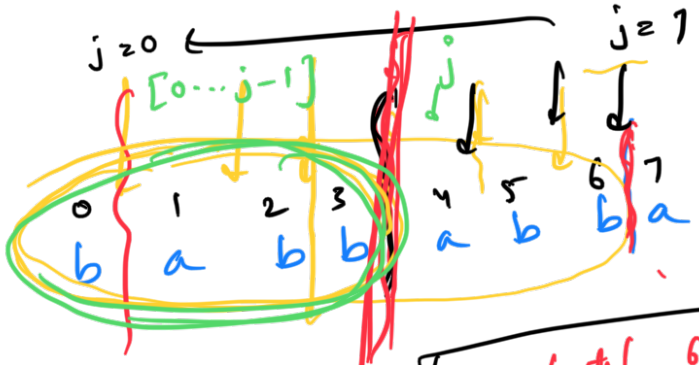
Approach 1: Wrong



(b, aa, bbb) → 2 ✗

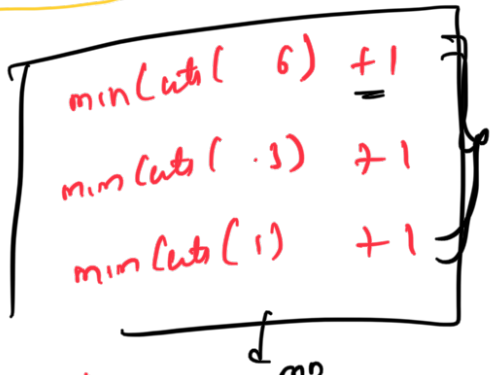
(ba)

Approach 2:



s =

dp[i] = (0...i)



min

0 1 2 3 4

s = a b c d e

minCuts(0) → 'a'

ind = 0

(i.... end)

```
if(ind == -1)
    return -1;

int minCuts(int ind) {
    if(ind == -1 || ind == 0) return 0;
    if(isPalindrome(0...ind)) return 0;
    ans = INT_MAX;
    for(j = ind; j >= 0; j--) {
        if(isPalindrome(j, ind)) {
            ans = min(1 + minCuts(j-1), ans);
        }
    }
    return ans;
}
```

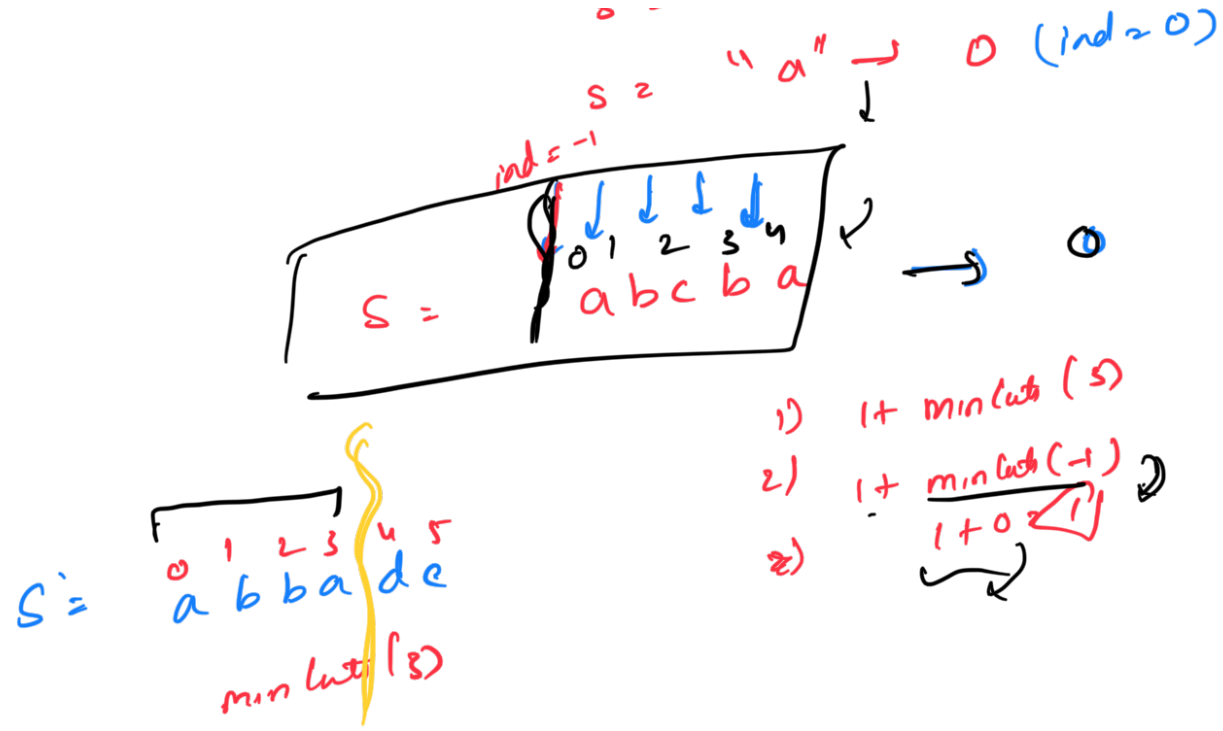
O(n²)

O(n)

O(n)

O(n)

s = "" → 0 → ind = -1



minCuts (abba)

T.C: O (# states × Tc per state)

O (n × O(n²))

⇒ O(n³) → O(n²)

Question: Palindromic Substring Queries

s = 0 1 2 3 4
b b c b b [N]

Queries (i, j)

Q = 10^6

n = 10^3

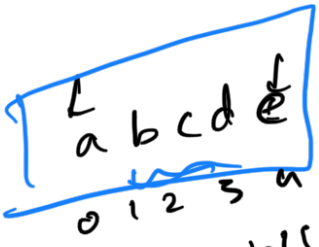
(1, 3) → ✓

(0, 3) → ✗

(0, 4) → ✓

Brute force:

T.C.: O(n × n)



dp[0][4] = false

s[i:j]
dp[i][j] = false

Approaches:

s[i ..... j] →

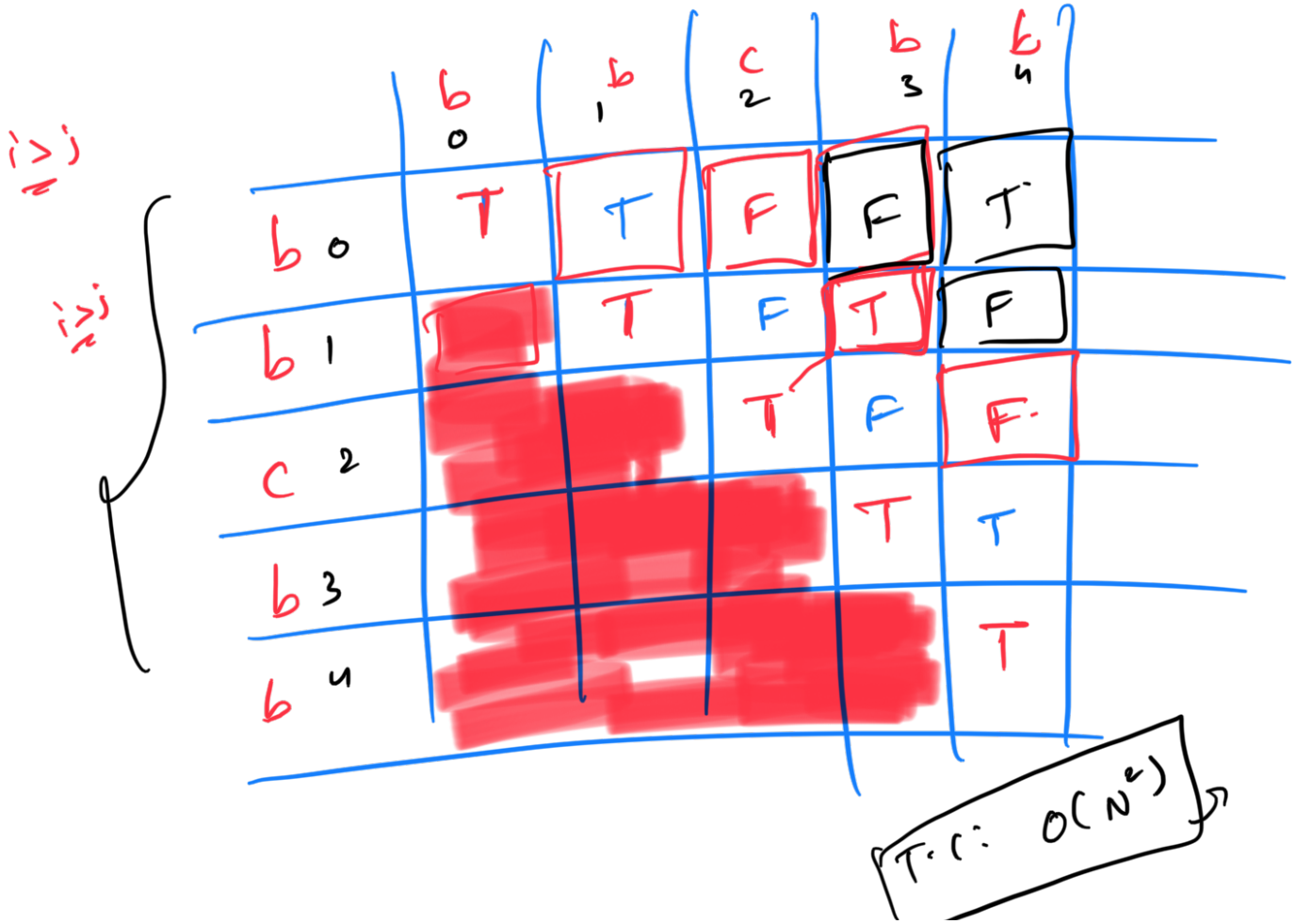
if (s[i] == s[j] && isPalindrome(i+1, j-1))

isPal[i][j] = true;

else
isPal[i][j] remain false.

dp[i][j] = ?

[0...4]
b b



T.C.: O(N²)

Question: Coin Sum Infinite

C = 1 3 4

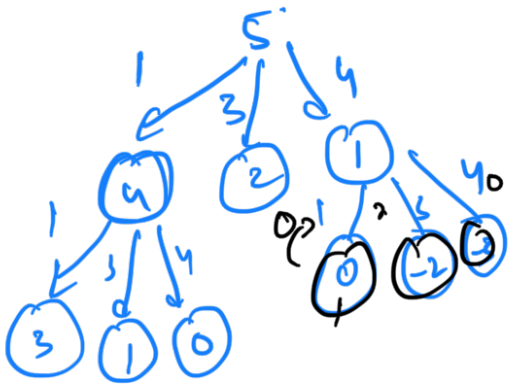
T = 5

{1,1,1,1,1} {1,1,3} {1,4}

Case1: Order matters

{1,1,1,1,1} {1,1,3} {1,3,1} {3,1,1}

{1,4} {4,1} => 6 ways



dp[T]

```
int ways (A[], T ) {
    if (T == 0) return 1;
    if (T < 0) return 0;
    ans = 0
    for (i=0; i < A.length; i++) {
        ans += ways ( A, T - A[i] );
    }
    return ans;
}
```

O(A.length)

MAX

T·C per state

T.C: O([illegible]

↓

T.C: O(T x N)

S.C: O(T)

→ Case 2: order Does not Matter

A = 1 3 5

T = 6

{1,1,1,1,1,1} {1,1,1,3} {1,5}

{3,3}

choices for ith coin

1) Don't take it

2) Take one or more

A = 1 3 5 X

T = 6

→ ways(6, [1,3])

2) ways(1, [1,3,5])

ways(T, i) = ways(T, i-1) +

(T, A[i], i);

```
dp[3][]

int ways(int T, int i){
    if(T<0) return 0;
    if(T==0) return 1;
    if(i<0) return 0;

    ans += ways(T, i-1);
    ans += ways(T-A[i], i);

    return ans;
}
```

T.C: $O(\#States \times T.C\ per\ call)$

$T \times N \times O(1)$

T.C: $O(T \times N)$

S.C: $O(T \times N)$

$O(T)$

Bottom-Up

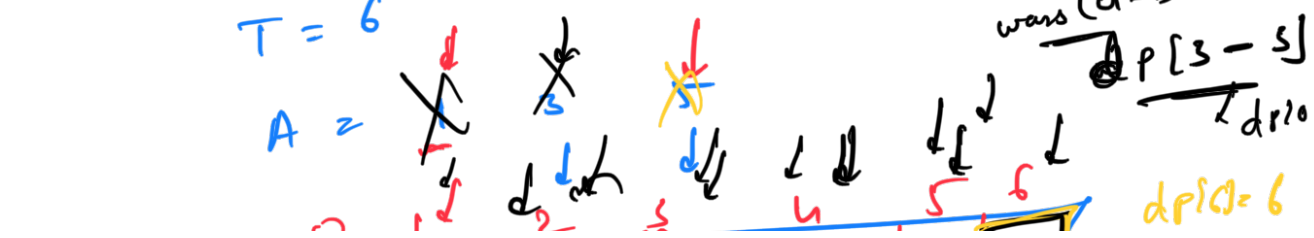
dp[T]

dp[i] => No. of ways to make a target 'i' using the coins

dp[0] = 1

T = 6

A =



dp[] =

dp[T - coins[i]]

target = 5
coins = 1
5 - 1 = 4

```
dp[0] = 1;
for(j = 0; j < coins.length(); j++){
    for(t = 1; t <= T; t++){
        if( t - coins[j] >= 0){
            dp[t] = dp[t] + dp[t - coins[j]]
        }
    }
}
```

?

T.C: O(T x N)

S.C: O(T)

S = a b b a

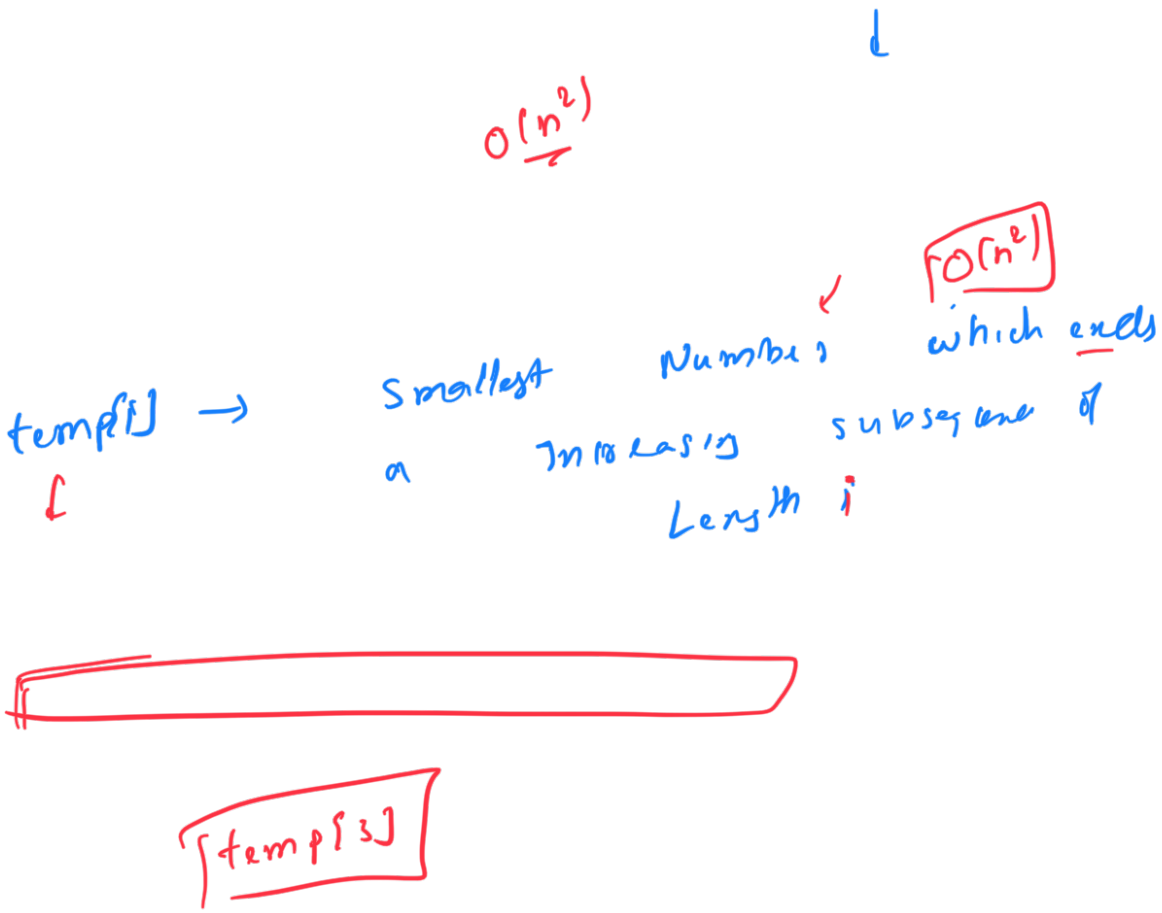
P = a

X

A = bba

Case 1 : (abba, a)

Case 2 :



Wildcard Pattern Matching : Top-Down

```
dp[s.length()+1][p.length() + 1] = {-1};
bool isMatch(int ls, int lp){
      if(ls == 0 && lp == 0) return true;
      if(lp == 0) return false;
      if(ls == 0){
            for(int i = 0; i < lp; i++)
                  if(p[i] != '*') return false;
            return true;
      }
      if(dp[ls][lp] != -1) return dp[ls][lp];
      if(s[ls - 1] == p[lp - 1] || p[lp - 1]=='?')
            dp[ls][lp] = isMatch(ls - 1, lp - 1);
      else if (p[lp - 1]=='*'){
            dp[ls][lp] = isMatch(ls - 1, lp) || isMatch(ls, lp - 1);
      }
      else
            dp[ls][lp] = false;
      return dp[ls][lp];
}
```

Coin Sum : Top-Down [Duplicates allowed]

```
int ways(A[], int T){
      if(T < 0) return 0;
      if(T == 0) return 1;
      int ans = 0;
      for(int i = 0; i < A.length(); i++){
            ans += ways(A, T - A[i]);
      }
return ans;
}
```

Coin Sum : Bottom-Up [Duplicates allowed]

```
for(i = 1; i<=T; i++){
      for(int j = 0; j < A.length(); j++){
            if(i - A[j] >= 0)
                  ways[i] += ways[i - A[j]]
      }
}
return ways[n];
```

Coin Sum :Top-Down [Duplicates not allowed]

```
dp[T+1][N] = {-1};
int ways(A[], int T, int n){
      if(T < 0) return 0;
      if(T == 0) return 1;
      if(n < 0) return 0;

      if(dp[T][n] != -1) return dp[T][n];
      int ans = 0;
      ans += ways(A, T, n -1);
      ans += ways(A, T - A[i], n);
      dp[T][n] = ans;
      return dp[T][n];
}
```

LPS : Top - Down:

```
dp[N][N] = {-1};
int lps(string s, int i, int j){
      if(i > j) return 0;
      if(i == j) return 1;
      if(dp[i][j] != -1) return dp[i][j];
      if(A[i] == A[j])
            dp[i][j] = 2 + lps(s, i + 1, j - 1);
      else
            dp[i][j] =  max(lps(s, i+i, j), lps(s, i, j - 1);
return dp[i][j];
}
```

Coin Sum :Bottom Up [Duplicates not allowed]

```
for(int j = 0; j < A.length(); j++){
      for(i = 1; i<=T; i++){
            if(i - A[j] >= 0)
                  ways[i] += ways[i - A[j]]
      }
}
return ways[n];
```

LPS : Bottom-up

```
dp[N][N] = {0};
for(int i = 0; i < s.length(); i++) dp[i][i] = 1;
for(int len = 2; len <= s.length(); len++){
      for(int i = 0; i <= s.length() - len; i++){
            int j = i + len - 1;
            if(s[i] == s[j]
                  dp[i][j] = 2 + dp[i + 1][j - 1];
            else dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
      }
}
return dp[0][n-1];
```

Longest Palindromic Substring : Bottom-up

```
dp[N][N] = {false};
for(int i = 0; i < s.length(); i++) dp[i][i] = true;
for(int len = 2; len <= s.length(); len++){
      for(int i = 0; i <= s.length() - len; i++){
            int j = i + len - 1;
            if(s[i] == s[j]
                  dp[i][j] = dp[i + 1][j - 1];
            else
                  dp[i][j] = false;
      }
}
return dp[0][n-1];
```

Palindromic Partition

```
int minCut(int ind){
      if(ind < 0 || isPalindrome(0, ind))return 0;

      ans = INT_MAX;
      for(j = ind; j >= 0; j-){
            if(isPalindrome(j, ind)){
                  ans = min(ans, 1 + minCut(j-1));
            }
      }
      return ans;
}
```