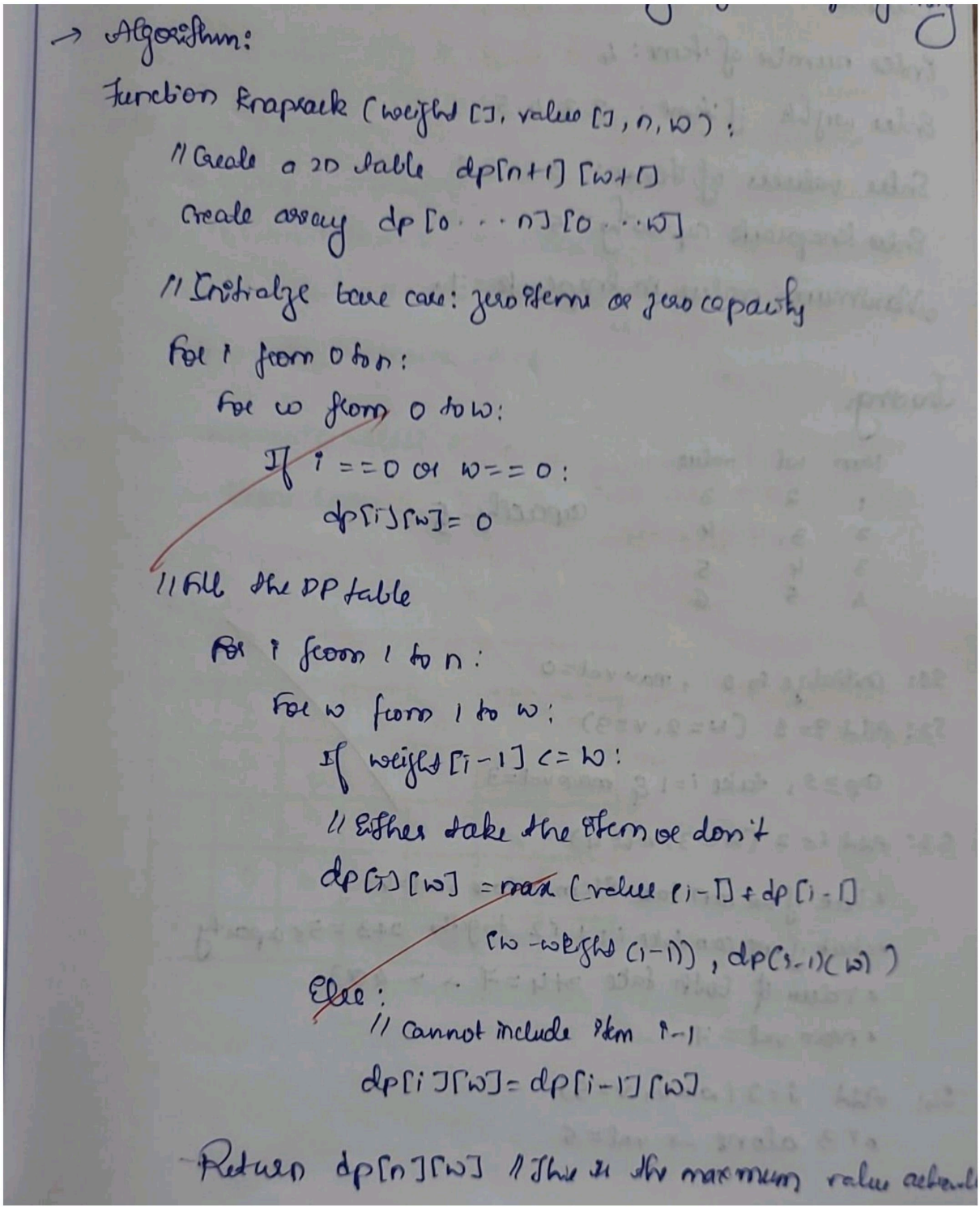
PROGRAM 5

Implement 0/1 Knapsack problem using dynamic programming.

ALGORITHM



→ Algorithm:

Function Knapsack (weight [], value [], n, W):

// Create a 2D table dp[n+1] [W+1]

Create array dp [0 . . . n] [0 . . . W]

// Initialize base case: zero items or zero capacity

For i from 0 to n:

For w from 0 to W:

If i == 0 or w == 0:

dp[i][w] = 0

// Fill the DP table

For i from 1 to n:

For w from 1 to W:

If weight[i-1] <= W:

// Either take the item or don't

dp[i][w] = max (value[i-1] + dp[i-1]

[w - weight(i-1)], dp(i-1)(w))

Else:

// Cannot include item i-1

dp[i][w] = dp[i-1][w]

Return dp[n][w] // This is the maximum value obtain

CODE

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve knapsack problem using DP
int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];

    // Build table dp[][] in bottom-up manner
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][W];
}

int main() {
    int n, W;

    printf("Enter number of items: ");
```

```
    scanf("%d", &n);

    int wt[n], val[n];

    printf("Enter weights of items:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &wt[i]);

    printf("Enter values of items:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &val[i]);

    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    int maxValue = knapsack(W, wt, val, n);

    printf("Maximum value that can be obtained = %d\n", maxValue);

    return 0;
}
```

OUTPUT

```
Enter number of items: 4
Enter weights of items:
2 3 4 5
Enter values of items:
3 4 5 6
Enter capacity of knapsack: 5
Maximum value that can be obtained = 7
```

TRACING

Tracing:

Item	wt	value
1	2	3
2	3	4
3	4	5
4	5	6

capacity - 5

S1: Initialize to 0 , max val = 0

S2: Add i = 1 (w = 2, v = 3)

cap ≥ 2, take i = 1 & max val = 3

S3: Add i = 2 (w = 3, val = 4)

• check if we can take Item 2 alone → val = 4

• check if we can take i1 + i2 together 2+3 = 5 ≤ capacity

• value if both take 3+4 = 7 → > 4 & 3.

• max val = 7.

S4: Add j = 3 (w = 4, v = 5)

• i3 alone → val = 5

• i1 + i3 → wt = 2+4 = 6 ≤ capacity - yes

• i2 + i3 → wt = 3+4 = 7 ≤ cap → no

So max value stays the same 7.

S5: Add j = 4 (w = 5, v = 6)

• i = 4 alone → v = 6.

i3 + i4 → wt = 2+5 = 7 < capacity — no

i2 + i4 → wt = 3+5 = 8 < capacity — no

max value remains same

So Final maximum value = 7

Items Chosen 1 & 2 (wt 2+3 = 5).

Val 3+4 = 7

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Back tracking

- i=4, w=5
- dp[4][5] = dp[3][5] = 7

  4 not included.

  w=5 i=3
- dp[3][5]=7 } equal
  dp[2][5]=7

  3 not included

  w=5 i=2
- dp[2][5]=7 } no equal
  dp[1][5]=3

  2 included

  w = 5 - wt[2-1] = 5-3 = 2.

  i=1
- dp[1][2]= 3 ≠ dp[0][2] = 0

  Item 1 was included.

  w = 2 - w[1-1] = 2-2 = 0

⇒ Total weight
= 2+3 = 5

Total value
= 3+4 = 7.

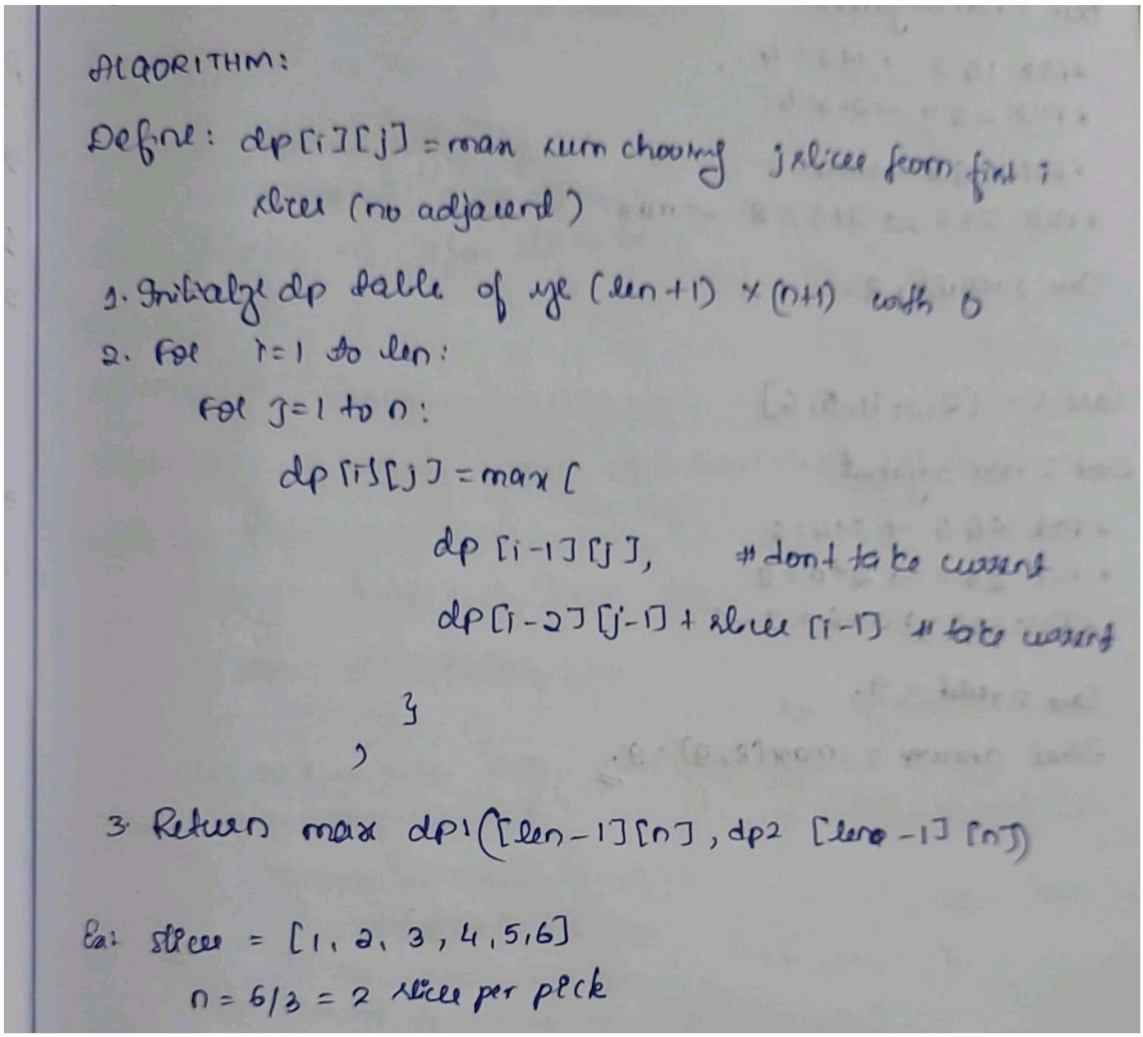
Sahil 17/8/25

6/6

LEETCODE 4

PIZZA WITH 3n SLICES

ALGORITHM



CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
// Function to compute the max sum using DP (like 0/1 Knapsack)
int calculate(int* slices, int start, int end, int n) {
    int len = end - start + 1;
    int dp[len + 2][n + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = len - 1; i >= 0; i--) {
        for (int j = 1; j <= n; j++) {
            int take = slices[start + i] + dp[i + 2][j - 1];  // pick slice
            int skip = dp[i + 1][j];                          // skip slice
            dp[i][j] = max(take, skip);
        }
    }
    return dp[0][n];
}

int maxSizeSlices(int* slices, int slicesSize) {
    int n = slicesSize / 3;
    int max1 = calculate(slices, 0, slicesSize - 2, n);     // exclude last
    int max2 = calculate(slices, 1, slicesSize - 1, n);     // exclude first
    return max(max1, max2);
}
```

OUTPUT

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

slices =

[1,2,3,4,5,6]

Output

10

Expected

10

Contribute a testcase

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

slices =

[8,9,8,6,1,1]

Output

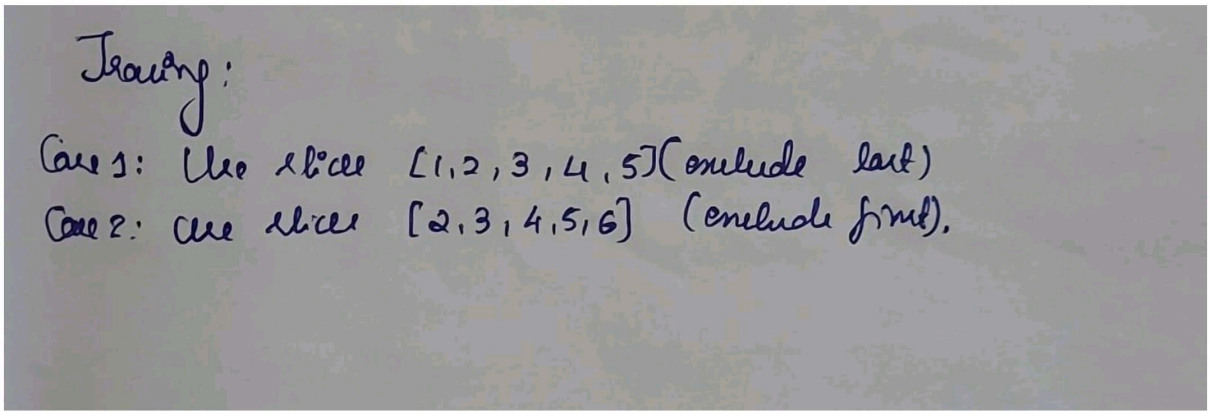
16

Expected

16

Contribute a testcase

TRACING



Case 1: [1, 2, 3, 4, 5]

Best 2 non adjacent

* Pick 1 & 3 → 1+3 = 4
* Pick 2 & 4 → 2+4 = 6
* Pick 1 & 4 → 1+4 = 5
* Pick 3 & 5 → 3+5 = 8 ✓ max.

Case 1 result = 8.

Case 2 : [2, 3, 4, 5, 6]

Best 2 non adjacent

* Pick 2 & 4 → 2+4 = 6
* Pick 2 & 5 → 2+6 = 8
* Pick 3 & 5 → 3+6 = 9 ✓

Case 2 result = 9.

Final answer = max(8, 9) = 9.