Q1. Bottom-Up DP #1

Comparing Bottom-Up dp with top down dp choose the correct option :

bottom up dp codes execute considerably faster than top down codes.

we can avoid the overhead function calls in bottom-up dp.

All of the above.

It is much easier to analyse time and space complexity of a bottom up code.

Q2. Bottom-Up DP #2

Which of the following is the correct time and space complexity respectively to find the N^{th} fibonnaci number using bottom-up DP :

O(N*N), O(N)

O(N), O(1)

O(N), O(N)

O(1), O(N)

Q3. Bottom-Up DP #3

Which of the following technique helps calculate the N^{th} fibonacci number in O(logN) time:

SOS DP

It is not possible to do it in O(logN)

DP with bitmasks

Matrix Exponentiation

Q4. Bottom-Up DP #4

While finding the sum of the max sum subarray in an array A[1..N], we define dp[i] = sum of maximum sum subarray that ends at the i^{th} element of the array and starts on or before it. Choose the correct recurrence for i > 1 and i <= N (1-based indexing)

dp[i] = dp[i-1] + dp[i-2]

dp[i] = max(A[i], dp[i-1] + A[i-1])

dp[i] = max(dp[i-1] + A[i], A[i])

dp[i] = max(A[i] + A[i-2], A[i])

Q5. Bottom-Up DP #5

Choose the best possible time and space complexity respectively for finding the sum of the maximum sum subarray :

O(N*N), O(N)

O(N), O(N)

O(1), O(N)

O(N), O(1)

Q6. Bottom-Up DP #6

For finding the sum of the max sum subarray in an array A[1..N], a student instead of using kadane's algorithm does the following :

for all i (2 to N) calculate:
best[i] = prefix*sum(i) - [minimum(prefix*sum(j)) over all j from 1 to i - 1].
output [max(best[j]) over all j from 1 to N].

where prefix_sum(i) = A[1] + A[2] + A[3] + … + A[i - 1] + A[i] and best[1] = A[1].

choose the correct option:

The algorithm is incorrect.

This algorithm is worse than kadane's algorithm.

The algorithm is correct and can be implemented in O(N) time and O(1) space.

The algorithm is correct and can be implemented in O(N) time but not O(1) space.

Q7. Bottom-Up DP #7

Given an array ARR of N elements [1..N], we wish to select two non-overlapping subarrays of this array such that the sum is maximum possible.

We come up with the algorithm:

```
select subarray 1 [i..j]
       select subarray 2[k..l] such that k > j
do for all possible i <= j < k <= l
```

what is the time complexity of the above algorithm?

N^3

N^2LogN

NlogN

N^4

Q8. Bottom-Up DP #8

Given an array ARR of N elements [1..N], we wish to select two non-overlapping subarrays of this array such that the sum is maximum possible.

We feel that kadane's algorithm can help to reduce the time complexity. So We come up with the algorithm:

```
choose i
      select best sum subarray in [1..i]
      select best sum subarray in [i+1..N]
for all valid i.
```

what is the time complexity of the above algorithm?

N^3LogN

N^4

N

N^2

Q9. Bottom-Up DP #9

Given an array ARR of N elements [1..N], we wish to select two non-overlapping subarrays of this array such that the sum is maximum possible.

We feel that running kadane's algorithm again and again is not needed and a better solution can be derived. The prev algorithm consisted of getting the sum of the best sum subarray in region 1..i and i+1..N

We define an array bestSumLeft such that bestSumLeft[i] represents the sum of the best sum subarray in [1..i] and similarly we define bestSumRight.

Assume that Kadane[i] stands for the max sum subarray ending at the ith element of the array. Also bestSumLeft[1] = ARR[1].

Which of the following is the correct recurrence equation for bestSumLeft :

bestSumLeft[i] = max(bestSumLeft[i-1], ARR[i])

bestSumLeft[i] = max(bestSumLeft[i-1] + ARR[i], Kadane[i])

bestSumLeft[i] = max(bestSumLeft[i-1], Kadane[i])

bestSumLeft[i] = max(bestSumLeft[i-1] + bestSumLeft[i-2], ARR[i])

Q10. Bottom-Up DP #10

Given an array ARR of N elements [1..N], we wish to select two non-overlapping subarrays of this array such that the sum is maximum possible.

Assuming that we can similarly build up the Array bestSumRight in Linear time, then what is the best time and space complexity respectively we can achieve to solve the problem?

N,N

N^2, N

N^2,N^2

N,N^2

Q11. Bottom-Up DP #11

Given the expression **A1 opr1 A2 opr2 …. opr(N-1) AN** where A_i is an integer and opr_i is either + or - for example : 5 + 3 - 8 + 6.

Put appropriate parenthesis so that the value of the expression becomes maximum.

As in 5 + (3 - 8) + 6 = 6 is greater than (5 + 3) - (8 + 6) = -6.

The given problem is the most similar to which of the following classic DP problem :

Max-Sum Subarray problem

Matrix-Chain Multiplication

problem of finding the LIS

It is not a DP problem

Q12. Bottom-Up DP #12

Given the expression **A1 opr1 A2 opr2 …. opr(N-1) AN** where A_i is an integer and opr_i is either + or - for example : 5 + 3 - 8 + 6. Put appropriate parenthesis so that the value of the expression becomes maximum. As in 5 + (3 - 8) + 6 = 6 is greater than (5 + 3) - (8 + 6) = -6.

Let us store operands in vector OP and operators in vector OPR. Let dp[i][j] represent maximum possible value of the subexpression **Ai opr[i] Ai+1 opr[i+1] Ai+2 … opr[j-1] Aj.**

choose the correct expression :

dp[i][j] = maximum possible value of { dp[i][k] OPR[k] dp[k+1][j] } over all K from i to j-1

dp[i][j] = Ai OPR[i] Ai+1 OPR[i+1] Ai+2 … OPR[j-1] Aj

dp[i][j] = maximum possible value of { dp[i][k] + dp[k+1][j] } over all K from i to j-1

It is not a DP problem.

Q13. Bottom-Up DP #13

For an array ARR of N elements what is the best possible known time complexity in which we can find the length of the longest increasing subsequence.

NlogN

N^2

N

Nlog^2N

Q14. Bottom-Up DP #14

Instead of coding up the algorithm for LIS, a student does the following:

```
Creates a copy of ARR.
Sorts the copy.
Finds length of longest Common subsequence of ARR and sorted copy.
```

is the student correct?

only for some of the inputs.

yes the student is correct.

the solution is incorrect for all inputs.

this is not the right approach and one must always memorize the standard techniques

Q15. Bottom-Up DP #15

You wish to find the length of the longest bitonic subsequence in the array ARR[1..N].

Since you know LIS problem, you are able to calculate the arrays LIS and LDS, where LIS[i] represents length of longest increasing subsequence that ends at i and starts on or before i.

LDS[i] represents length of longest decreasing subsequence that starts at i and ends on or after i.

Which of the following correctly evaluate the length of the longest bitonic subsequence:

max of LIS[i] + LDS[i] - 1 for all i.

max of LIS[i] + LDS[i] for all i.

min of LIS[i] + LDS[i] + 1 for all i

it is an NP-hard problem.