1. $Maximum\ sum\ in\ binary\ tree$

$as\ the\ constraint\ would\ be\ adjacent\ nodes\ cannot\ be\ picked$

$so\ each\ time\ iterating\ the\ binary\ tree,\ the\ optimal\ solution\ would\ be\ take\ that\ node\ or\ not\ take\ it$

$if\ we\ take\ that\ node,\ we\ must\ take\ the\ grandchildrens\ in\ order\ to\ maximize\ the\ sum$

$if\ we\ don't\ take\ that\ node,\ we\ must\ take\ the\ both\ childrens\ since\ they\ are\ not\ adjacent$

$and\ the\ condition\ of\ taking\ the\ node\ or\ not\ would\ be$

$Let\ A\ be\ the\ binary\ tree\ with\ A[0]\ is\ the\ root\ node$

$max((A[0]\ +\ A[3]\ +\ A[4]\ +\ A[5]\ +\ A[6]), A[1]\ +\ A[2])$

$and\ this\ would\ be\ the\ simplest\ form\ to\ solve\ this\ kind\ of\ problem$

$extend\ to\ solving\ the\ general\ problem$

$Let\ dp[n]\ to\ be\ the\ array\ storing\ the\ maximum\ of\ n\ node\ can\ get\ initialize\ all\ 0$

$function\ maxSum(node)$

$base\ case\ if\ no\ node\ then\ return$

$if\ dp[index\ of\ node]\ not\ equal\ to\ zero\ means\ this\ node\ has\ been\ get\ the\ optimal\ solution\ so\ return$

$max\ sum\ from\ left\ left\ child\ =\ maxSum(node.left.left)$

$max\ sum\ from\ left\ right\ child\ =\ maxSum(node.left.right)$

$max\ sum\ from\ right\ left\ child\ =\ maxSum(node.right.left)$

$max\ sum\ from\ right\ right\ child\ =\ maxSum(node.right.right)$

$max\ sum\ from\ grandchildren\ =\ (maxSum(node.left.left)+ maxSum(node.left.right)+maxSum(node.right.left)+maxSum(node.right.right))$

$max\ sum\ from\ left\ child\ =\ maxSum(node.left)$

$max\ sum\ from\ right\ child\ =\ maxSum(node.right)$

$since\ we\ either\ choose\ grandchildren\ or\ both\ children\ so\ the\ return\ would\ be\ maximum\ between\ them$

$adding\ the\ optimal\ solution\ in\ dp[index\ of\ node]$

$return\ max((node\ +\ max\ sum\ from\ grandchildren),\ (max\ sum\ from\ left\ child\ +\ max\ sum\ from\ right\ child))$

$The\ initial\ call\ would\ be\ maxSum(A[0])$

$since\ this\ algorithm\ will\ iterate\ each\ node\ once\ so\ the\ time\ complexity\ would\ easily\ be\ taken\ as\ O(n)$

2. $Longest\ increasing\ path\ in\ 2D\ Matrix$

$since\ every\ time\ the\ path\ can\ only\ go\ downward\ or\ rightward$

$for\ the\ optimal\ solution\ if\ M[i, j] < M[i + 1, j]\ or\ M[i, j] < M[i, j + 1]$

$then\ M[i, j]\ must\ include\ in\ the\ solution\ and\ the\ largest\ path\ accumulated\ from\ right\ or\ down\ plus\ 1$

$so\ the\ algorithm\ just\ need\ to\ recursively\ doing\ this\ condition\ checking$

$Let\ dp[n][n]\ initialized\ as\ same\ size\ with\ M\ and\ all\ element\ to\ be\ 1$

$global\ variable\ Longest = 0$

$define\ a\ function\ LIP(inputting\ i, j)$

$boundary\ and\ base\ case\ check$

$if\ i > n\ or\ j > n\ then\ return$

$if\ dp[i][j] != 1\ then\ this\ cell\ has\ been\ checked\ so\ return$

$recursively\ call\ function$

$LIP(i + 1, j)$

$LIP(i, j + 1)$

$if\ i + 1 < n\ and\ M[i][j] < M[i + 1][j]$

$save\ the\ optimal\ solution\ in\ dp[i][j]$

$dp[i][j] = max(dp[i][j], dp[i + 1][j]) + 1$

$if\ j + 1 < n\ and\ M[i][j] < M[i][j + 1]$

$dp[i][j] = max(dp[i][j], dp[i][j + 1]) + 1$

$compare\ the\ longest\ to\ dp[i][j]\ to\ check\ whether\ dp[i][j]\ have\ a\ better\ length\ then\ longest$

$longest = max(longest, dp[i][j])$

$the\ initial\ call\ would\ be\ LIP(0, 0)$

$after\ running\ the\ whole\ LIP\ function\ call$

$global\ variable\ longest\ would\ be\ the\ longest\ path$

$For\ the\ time\ complexity$

$this\ LIP\ function\ recursively\ run\ over\ each\ element\ from\ M\ once$

$as\ if\ dp[i][j] != 1\ would\ stop\ the\ LIP\ function\ from\ calling\ the\ checking\ element\ again$

$and\ each\ time\ of\ recursion\ would\ do\ constant\ time\ of\ comparison$

$such\ that\ the\ LIP\ function\ would\ run\ in\ O(n^2)\ since\ there\ are\ n^2\ elements$

$For\ the\ space\ complexity$

$this\ algorithm\ require\ a\ array\ ap\ same\ size\ as\ M\ n\ by\ n\ and\ one\ more\ global\ variable\ longest$

$so\ the\ space\ complexity\ would\ be\ n^2 + 1 = O(n^2)$

3. *Set partitioning*

1. *this question can be understand as if there is a subset of S have half of sum of S*

let $dp[i][j]$ *be a array storing if there is a subset of* $S[1\ to\ n]$ *have sum to i*
at each time determining if there is a subset equal to i
we can divide it into two situations
1. *there is a subset of* $S[1, j-1]$ *excluding* $S[j]$ *have sum to i*
2. *the sum i can be obtained from a subset of* $S[1, j-1] + S[j] = i$

if both case is false means the set $S[1\ to\ j-1]$ *cannot have a sum equals to i*

$Let\ sum = sum(S)/2$
initialize all the part of $dp[0][0\ to\ n] = true$ *as 0 can be get from empty set*
initialize all the part of $dp[1\ to\ n][0] = false$ *as empty set cannot have sum* > 0

$for\ i = 1\ to\ sum$
    $for\ j = 1\ to\ n$
        inherit the previous subset of not including element $S[j]$
        $dp[i][j] = dp[i][j-1]$
        $if\ i >= S[j]$
        means the sum i can be achieved by subset of $S[1, j-1]$ *and* $S[j]$ *otherwise it is impossible*
        because any subset adding $S[j]$ *would larger than i*
        then either $dp[i - S[j]][j-1]$ *is true or it follow the* $dp[i][j-1]$

after all the iteration of set S, the $dp[sum][n]$ *indicate whether there is a subset of S have half of sum of S*
and this will be the answer of this question

For the time complexity
*there are sum * n times iterations which is equals to* $O(sum * n)$
for the space complexity
the dp array is created to store every subset $S[0\ to\ n]$ *to* $[0\ to\ sum]$ *solutions, so it is* $O(sum * n)$

2.
to solve this problem, the algorithm would reuse the algorithm from 1. to build the dp array same as 1.
then create a new iteration as now $dp[sum][n]$ *may not be true*
so the iteration would keep checking on the largest $dp[i]$ *to be true as this would make the difference to be smallest*
$Let\ sum = Sum(S)//2$
$for\ i = sum\ to\ 0$
    if $dp[i][n]$ *is true then return and we know i would be the smallest number to sum*

then after returning $dp[i][n]$ *would be answer*

for time complexity since it is the same as iterating all the element and matching with 1 to sum
so it is the same as 1. $O(sum * n)$
for space complexity it also uses the dp array to store all the conditions, it is also the same as the 1. to be $O(sum * n$

4. *Continuous subarray partitioning*

the recurrence for the optimal solution would be each time take a subarray, then search the remaining array
each time $m = m - 1$ *doing the slicing and returning the* $max(Sum(subarray), Sum(remaining\ array))$
then try all the subarray and find the minimum of all the $max(Sum(subarray)$
$, Sum(remaining\ array))$
this is a brutal force to check all the subarray that A divided m times

for introducing the dp solution algorithm need a 2d array $dp[n][m]$ *recording for n size subset with*
slicing m times condition and we can use it to deduce next element's situation
first of all need a array S implemented as follow

$S[1] = A[1]$
$For\ i = 2\ to\ len(A)$
$\quad S[i] = A[i] + S[i - 1]$

and we can easily get the sum of subarray and remain array then use this to do the comparison
so each time on a element the algorithm should find the all possible outcomes from all the slicing and find the min
Let i denoted as i^{th} *element and j denoted as* j^{th} *slice*

$dp[i][j] = min(dp[i][j], max(dp[k][j - 1], S[i] - S[k]));$

$dp[k][j - 1]$ *is indicating the subarray's sum*
$S[i] - S[k]$ *means the remaining array's sum*
the k is looping over from 1 to i to get the minimum of all the possible slicing for subarray of $A[1\ to\ i]$

so the whole implementation would be

$for\ i = 1\ to\ n$
$\quad dp[i][1] = S[i]$ *as doing no slicing the maximum sum would the the accumulating sum* $S[i]$
$For\ i = 1\ to\ n$
$\quad For\ j = 2\ to\ m$
$\quad\quad For\ k = 1\ to\ i$
$\quad\quad\quad dp[i][j] = min(dp[i][j], max(dp[k][j - 1], S[i] - S[k]));$

then the value storing in $dp[n][m]$ *would be the answer*

for time complexity
this algorithm mainly run through n elements and m times partitioning
and run in k times recording each time condition
since the worst case for k is n times
so the whole algorithm run in $O(n^2 * m)$ *time*

for space complexity
this algorithm created a extra dp array of size n x m so it takes $O(n * m)$ *space*