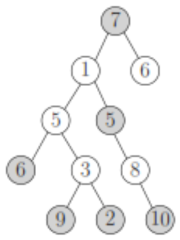
**Question 1 (20%): Maximum sum in binary trees**

Given a binary tree where each node has a positive number, describe a dynamic programming algorithm to select the subset of nodes with the maximum sum provided that adjacent nodes cannot be picked. In the following example, the shaded nodes form the optimal solution. For full credits, your algorithm should run in O($n$) time, where $n$ is the number of nodes. The algorithm just needs to output the sum, not the actual nodes picked.



**Solution**

Let $v(u)$ be the number of node $u$. Define $s(u)$ to be the maximum sum achievable for picking nodes from the subtree below (and including) $u$. Then

$$s(u) = \max\begin{cases} s(u.left) + s(u.right) \\ v(u) + s(u.left.left) + s(u.left.right) + s(u.right.left) + s(u.right.right) \end{cases}$$

Reasoning:

Case 1 means that we do not take *u* in the sum so that we take its children

Case 2 means that we take *u* in the sum so that we take its grandchildren

I use an array *S*[*u*], to store $s(u)$, so that I do not compute the same value multiple times. The following pseudocode is based on *memorization*.

```
Initialize $S[u]$=0 for every node $u$
MaxS ($root$)
---------------------------------------------------------------
MaxS($u$)
If $u$ is empty Return 0; // node does not exist
If $S[u]$ <>0 Return $S[u]$ // return stored value computed before
Swithout=0; // sum for case 1
Swith=0; // sum for case 2
Swithout= MaxS($u.left$) + MaxS($u.right$)
Swith= $v(u)$ + MaxS($u.left.left$) + MaxS($u.left.right$)+ MaxS($u.right.left$) + MaxS($u.right.right$)
If Swithout > Swith  $S[u]$= Swithout
        Else [$u$]=Swith
Return $S$ [$u$]
```

Alternative bottom up solution

I use post-order tree traversal, so that when I need the maximum sum for a node, the maximum sum for each node in its subtree has been computed. Thus, there is no need for recursive calls.

```
MaxSumTree($root$)
For each node $u$ in post-order traversal // the last node will be $root$
If $u$ is a leaf node
   $S[u]$=$v(u)$
   $S[u.left]$=0; $S[u.right]$=0 //needed to avoid references to empty cells of $S$
 Else // $u$ is an intermediate node
        Swithout= $S[u.left]$ + $S[u.right]$
        Swith=$v(u)$ + $S[u.left.left]$ + $S[u.left.right]$ + $S[u.right.left]$+ $S[u.right.right]$
        If Swithout > Swith, $S[u]$= Swithout
                else $S[u]$= Swith
Return $S[u]$
```

**Question 2 (20%): Longest increasing path in 2D Matrix**
Consider a *n*x*n* matrix *M* of distinct integers. We wish to find the longest path in *M* such that: cell *M*[*i*,*j*] can be reached by moving right from *M*[*i*,*j*-1], or down from *M*[*i*-1,*j*], provided that *M*[*i*,*j*-1]< *M*[*i*,*j*] or *M*[*i*-1,*j*] ] < *M*[*i*,*j*]. The path can start and end in any cell of *M*. Describe a dynamic programming algorithm that outputs the longest path, and discuss its time/space complexity.

Example: In the following matrix the longest paths are 1,3,5,6 and 2,4,7,8, both with length 4.

| *M* | *j*=1 | *j*=2 | *j*=3 |
|---|---|---|---|
| *i*=1 | 2 | 1 | 3 |
| *i*=2 | 4 | 7 | 5 |
| *i*=3 | 9 | 8 | 6 |

**Solution**
I use *L*[*i*,*j*] to denote the length of the longest path ending at *M*[*i*,*j*], and *Keep*[*i*,*j*] to remember the previous cell that I visited, before reaching *M*[*i*,*j*]
Set: *M*[*i*,0] =*M*[0,*j*]=*inf* and initialize *L*[*i*,*j*]=1
If *M*[*i*,*j*]> *M*[*i*,*j*-1] then *L*[*i*,*j*] =max{ *L*[*i*,*j*], *L*[*i*,*j*-1]+1}
If *M*[*i*,*j*]> *M*[*i*-1,*j*] then *L*[*i*,*j*] =max{ *L*[*i*,*j*], *L*[*i*-1,*j*]+1}

```
LIP($M, n, W$)
Initialize: $M[i,0] = inf$ for $0 \le i \le n$, $M[i,0] = inf$ for $0 \le j \le n$, $L[i,j] = 1$ for $1 \le i \le n$, $1 \le j \le n$, $Keep[i,j] = 0$ for $1 \le i \le n$, $1 \le j \le n$
maxL=1 // longest path found so far
Li=1; Lj=1 // indices i,j for the last cell in the longest path found so far
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        If  $M[i,j] > M[i,j-1]$
            $L[i,j] = L[i,j-1]+1$ //first time I visit $M[i,j]$; $L[i,j]=1$
            $Keep[i,j]$ = "←"
            If $L[i,j]$>maxL
                maxL= $L[i,j]$; Li=$i$; Lj=$j$
        If  $M[i,j] > M[i-1,j]$
            If ($L[i-1,j]+1$) > $L[i,j]$ //second time I visit $M[i,j]$; need to check $L[i,j]$
                $L[i,j] = L[i-1,j]+1$
                $Keep[i,j]$ = "↑ "
                If $L[i,j]$>maxL
                    maxL= $L[i,j]$ ; Li=$i$; Lj=$j$
PrintLIP(Li,Lj)
```

---

Pseudocode for Printing

```
PrintLIP($i,j$)
If $Keep[i,j]$== 0 RETURN
If $Keep[i,j]$= = "←"
            PrintLIP($i,j-1$)
    else // $Keep[i,j]$= = "↑ "
            PrintLIP($i-1,j$)
Output($M[i,j]$) // print $M[i,j]$) after the recursive calls so that the path is printed from beginning to end
```

The running time and space complexity is $O(n^2)$ since I visit all cells of the matrix and each cell requires constant cost.

**Question 3 (30%): Set partitioning**
Let *S* be a set of *n* positive integers.

1] Describe a dynamic programming algorithm that decides whether it is possible to partition *S* into two subsets *S1* and *S2* that have equal sum. Assume that the sum of all elements in *S* is even. Describe the time and space complexity, considering that we only need the True/False answer, but not the sets *S1*, *S2*.

Example: if *S* = {3,5, 9, 1}, then the answer is True (*S1*={3,5,1} *S2*={9}). If *S* = {3,5, 11, 1}, the answer is False.

2] Describe a dynamic programming algorithm that partitions *S* into two sets *S1* and *S2* so that the absolute difference between their sums is minimum. Your algorithm should output *S1* and *S2*. Describe the time and space complexity.
Example: if *S* = {3,5,11,1}, the answer is *S1*={3,5,1}, *S2*={11}, or *S1*={11}, *S2*={3,5,1}

**Solution**
1] Let *SUM* be the sum of all elements in the set. Let $W = SUM/2$.
I will use a Boolean Matrix $B[0,..,n \text{ x } 0,..W]$.
$B[i,j]$ is TRUE if I can have a sum equal to $j$, using the first $i$ numbers.
Boundary conditions: $B[0,j]$=False, $B[0,0]$ and $B[i,0]$=True

The recurrence is: $B[i,j]=B[i\text{-}1,j]$ OR $B[i\text{-}1,j\text{-}n_i]$
$B[i\text{-}1,j]$ means I already have sum equal to $j$ without using $n_i$.
$B[i\text{-}1,j\text{-}n_i]$ means that I use $n_i$, so that I must have the sum $j\text{-}n_i$ using the first $i$-1 numbers.

The solution is at the cell $B[n,W]$. The time complexity is $n\text{x}W$ and the space complexity $n+W$ because I only need the previous row.

Below is a modified version, based on the subset sum problem, Lecture 17, slides 3-11. In addition, a matrix *Keep* remembers the items that I take in the sum. *Keep*$[i,j]$ means that I need element $s_i$ for sum $j$. Although not required for part 1, we need *Keep* for part 2 of the question. In the following pseudo-code $s_i$ represents the i-th element of the set *S*.

```
SubsetSum($S, n, W$)
Initialize: $A[i,0] = True$ for $0 \le i \le n$, $A[i,j] = False$ for $0 \le i \le n$, $1 \le j \le W$, $Keep[i,j] = False$ for $1 \le i \le n, 1 \le j \le W$
for $i = 1$ to $n$ do
    for $j = 1$ to $W$ do
            if $s_i > j$ then // $s_i$ exceeds sum and cannot be taken
                $A[i,j] = A[i-1,j]$
            else // $s_i \le j$
             if $A[i-1,j]$= =TRUE
                   $A[i,j]$= TRUE
             Else // $A[i-1,j]$= =FALSE
                   If $A[i-1,j-s_i] == True$
                   $A[i,j] = True$
                   $Keep[i,j] = True$ // I need $s_i$ for sum $j$
```

The running time and space complexity is $O(nW)$ since I need to fill the cells of matrices *A* and *Keep*, and each cell requires constant cost.

2] If the pseudocode of part 1 returns True, I can partition into 2 sets with identical sums. If not, I find the last TRUE value in the last row $A[n,W'] = True$, where $W' < W$, and this corresponds to the minimum difference. The space requirement including *Keep, is* $n$x$W$. The following pseudocode creates the subsets *S1* and *S2* with the minimum difference using the *Keep* matrix of part 1.

```
S1=empty; S2=S;
GenSubset($n, W'$)

GenSubset($i, j$)
If $i$= =0 OR $j$ = = 0 RETURN
If Keep[$i$,$j$] == True
        S1= S1 ∪{$s_i$}; S2= S2 -{$s_i$}
        GenSubset($i-1, j-s_i$)
    Else
        GenSubset($i-1, j$)
```

**Question 4 (30%): Continuous subarray partitioning**

Given an array $A$ of $n$ positive integers and an integer $m$ ($m$ is much smaller than $n$), we wish to split $A$ into $m$ **non-empty continuous** subarrays, so that the largest sum among these $m$ subarrays is minimized. Provide the recurrence for the optimal solution, and describe a dynamic programming algorithm and its time/space complexity.

Example: Assume that the input array is $A$=[7,2,5,10,8] and $m$ = 2. There are four ways to split $A$ into two subarrays: **(1)** [7],[2,5,10,8], **(2)** [7, 2], [5,10,8], **(3)** [7, 2, 5],[10,8], **(4)** [7, 2, 5, 10],[8]
The best way is to split it into [7,2,5] and [10,8] because the largest sum between the two subarrays is only 18.

**Solution**

Let $s[j] = \sum_{k=1}^{j} A[k]$

With this pre-sum array $s$, $\sum_{k=i}^{j} A[k] = s[j] - s[i-1]$ can be computed in $O(1)$ time

For $1 \le j \le n, j \le i \le m, d[i,j]$ is the optimal value of the subproblem that we split $A[1..j]$ into $i$ non-empty continuous subarrays so that the largest sum among these $i$ subarrays is minimized.

Base case: $d[1,j] = s[j]$ (no partition)

Recursive case: $d[i,j] = min_{i-1 \le k < j}(max(d[i-1,k], s[j] - s[k]))$

The last split can be at position $i-1$ to $j-1$. When the last split is at position $k$, this means we have split the array into $i-1$ subarrays for the first $k$ elements, and $d[i-1,k]$ stores the largest sum among these $i-1$ subarrays. The sum of the $i$-th subarray is $s[j] - s[k]$. So, $max(d[i-1,k], s[j] - s[k])$ is the largest sum among the $i$ subarrays subject to the last split of $A[1..j]$ is at position $k$. We simply try all possible $k$ and pick the minimum.

```
CSP($A, n, m$)
s[1]= A[1]
For j = 2 to n  s[j]= s[j-1]+ A[j] // compute pre-sum array s
d[1,1]= A[1]
For j=1 to n d[1,j]= s[j] // base case for m=1 (no partitioning)
For i=2 to m  // general case – number of partitions >=2
    For j=i to n        // we cannot split if there are fewer elements than partitions (when j<i)
        minS=inf
        For k=i-1 to n-1 // last subbaray is A[k+1,..,j]. A[1,..,k] has i-1 partitions
            kSum=max(d[i-1,k], s[j]-s[k])
            If kSum < minS
                minS=kSum; Keep[i,j]=k
OutputSubArrays(A,n,m)
```

-------------------------------------------------------

```
OutputSubArrays($A$,$j$,$i$)
While $i$>0 and $j$>0
        OutputSubArrays($A$,$j$-$Keep$[$i$,$j$],$i$-1)
        Print elements $A$[$Keep$[$i$,$j$]]+1 to $A$[$j$]
```

-----------------------------------------------------------

The running time is $O(n^2m)$ because there are $O(nm)$ subproblems (i.e., cells in the 2D array), and each subproblem can be computed in $O(n)$ time.