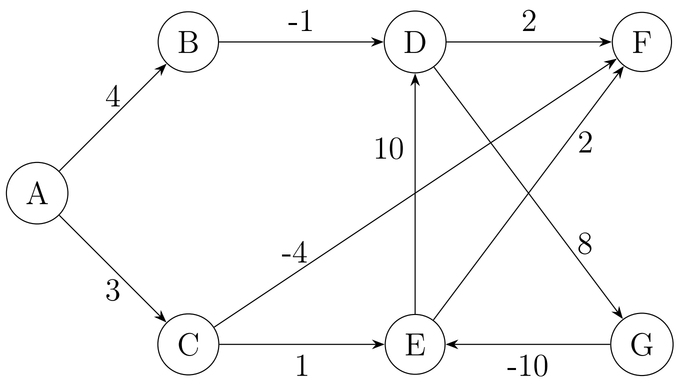
Assignment 3

Answers

Name: Pavlos Tomazos **ID:** 1115202200188
Name: Evangelos Argyropoulos **ID:** 1115202200010

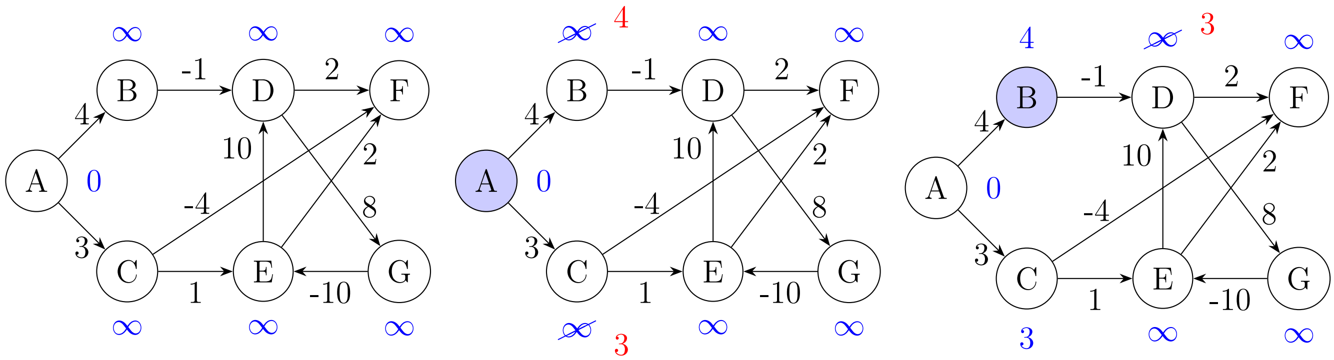
Exercise 1

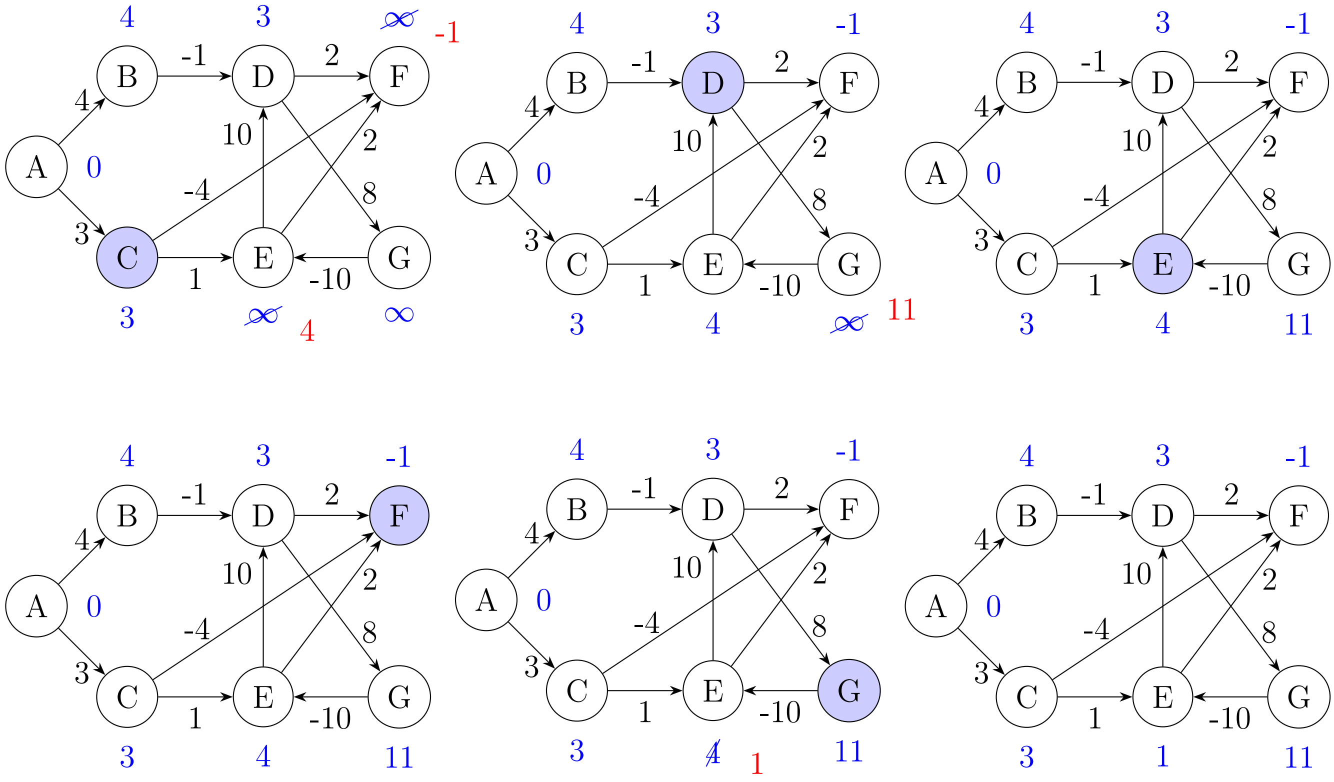
For the purpose of this exercise we will use the Academic ID: 1115202200188. Then the values are a = 1, b = 8, c= 8, resulting to the directed weighted graph below:



$E = \{(A,B),(A,C),(B,D),(C,E),(C,F),(D,G),(D,F),(E,D),(E,F),(G,E)\}$. The number of vertices is $|V| = 7$. So the iterations that the algorithm will perform to the edges is at most 6 (and one to check if there are negative cycles).

First iteration of the Bellman-Ford algorithm:





With a second iteration we can observe that no updates will occur, so we can terminate the algorithm.

Exercise 2

We suppose s is a string with length n. We can distinguish the following cases:

- n = 1: A string with only one character is trivialy palindromic
- n = 2: A string with two characters is palidromic iff the two characters are the same (for instance 'aa' is palindromic but 'as' is not)
- n >= 3: A string with at least 3 characters is palindromic iff the first and last characters are the same and the substring beetween those is also palindromic.

The following recursive propositional relation occurs:

$$sub(i,j) = \begin{cases} True & \text{, if } n = 1 \\ s_1 = s_2 & \text{, if } n = 2 \\ s_i = s_j \wedge sub(i+1, j-1) & \text{, if } n \geq 3 \end{cases}$$

For every (i,j), if the substring starting from i and ending at j is a palindrome, then the array at (i,j) will be **true**; otherwise, it will be **false**. By saving these results in an $n \times n$ array, we can easily retrieve useful information such as the starting and ending positions of the longest palindromic substring in s (LSB), or the length of the LSB (end $-$ start $+ 1$).

Algorithm 1 Find Longest Palindromic Substring of string s

```
1:  procedure LPS(s)        ▷ Let n be the size of string s and sub an array of size n × n
2:      start ← 0
3:      end ← 0
4:      for i = 1 to n do   ▷ Substrings that start and end with the same letter are palindromes
5:          sub[i, i] ← True
6:      end for
7:      for i = 1 to n − 1 do                            ▷ For substrings of length 2
8:          if s_i = s_{i+1} then                ▷ Check if the two characters are the same
9:              sub[i, i + 1] ← True
10:             start ← i
11:             end ← i + 1
12:         else
13:             sub[i, i + 1] ← False
14:         end if
15:     end for
16:     for l = 3 to n do                                ▷ For strings with length ≥ 3
17:         for i = 1 to n − l + 1 do              ▷ i is the starting index of the string
18:             j ← i + l − 1                                 ▷ j is the ending index
19:             if sub[i + 1][j − 1] = True and s[i] = s[j] then
20:                 sub[i][j] ← True
21:                 if j − i > end − start then
22:                     start ← i
23:                     end ← j
24:                 else
25:                     sub[i][j] ← False
26:                 end if
27:             end if
28:         end for
29:     end for
30:     return s[start to end], start
31: end procedure
```

The complexity of the provided algorithm is $O(n^2)$, as follows:

- In line 4, we have a loop with n steps, resulting in $O(n)$.
- In line 7, there is another $O(n)$ loop.
- Lines 16 and 17 contain nested loops with complexity $O(n) \cdot O(n) = O(n^2)$.

The final complexity is determined by the maximum of the complexities observed:

$$\max\{O(n), O(n), O(n^2)\} = O(n^2)$$

Exercise 3

(a) In the context of finding the Largest Independent Set (LIS) in a binary tree T, each node is represented as an element storing its data, children, and the LIS starting from it. To find the LIS for each node X:

- If X is a leaf node, its LIS is X itself.
- Otherwise, the LIS of X is the larger of:
  1. The LIS that includes X.
  2. The LIS that excludes X.

Based on the above and the known structure of a binary search tree, we can infer the following recursive relation:

$$\mathrm{LIS}(X) = \begin{cases} \{X\} & \text{, if } X \text{ is a leaf node} \\ \text{Larger}\left\{\{X\} \cup \left(\bigcup_{Y \text{ is a grandchild of } X} \mathrm{LIS}(Y)\right), \bigcup_{Y \text{ is a child of } X} \mathrm{LIS}(Y)\right\} & \text{, otherwise} \end{cases}$$

Algorithm 2 Compute the Largest Independent Set of a Binary Tree T rooted at *node*

```
procedure COMPUTE-LIS(node)
    if LIS of node is already computed then
        return LIS of node
    end if
    if node is a leaf then
        return set containing only the node
    end if
    LIS_excluding_node ← ∅
    for every child of the node do
        LIS_excluding_node add COMPUTE-LIS(child)
    end for
    LIS_including_node ← {node}
    for every grandchild of the node do
        LIS_excluding_node add COMPUTE-LIS(grandchild)
    end for
    if LIS_excluding_node has larger cardinality then
        return LIS_excluding_node
    else
        return LIS_including_node
    end if
end procedure
```

The provided algorithm computes the LIS of the whole tree by calling COMPUTE-LIS(root). The sub-problems are n (number of nodes) therefore the complexity is $O(n)$ as each sub-problem is solved only once.

(b) Below, the presentaion of the algorithm follows and the results are shown using an array. For each node in the tree we find the LIS of the subtree rooted in the node. When the size of the LIS excluding the current node is equal to the size of the LIS including it, we always choose the one that excludes the current node.

Initially, all the sets are empty.

root	A	B	C	D	E	F	G	H	I
LIS	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Then, firstly we determine the leaf nodes, because the algorithm trivialy sets the LIS of each leaf node as the node itself.

root	A	B	C	D	E	F	G	H	I
LIS	$\emptyset$	$\emptyset$	$\emptyset$	$\{D\}$	$\emptyset$	$\emptyset$	$\{G\}$	$\{H\}$	$\{I\}$

Then, knowing the LIS of the leaf nodes, we can compute the LIS of nodes E and F.

- $LIS(E) =$Larger$[E \cup \emptyset, LIS(G)] =$ Larger $[\{E\}, \{G\}] = \{G\}$
- $LIS(F) =$Larger$[\{F\} \cup \emptyset, LIS(H) \cup LIS(I)] =$ Larger$[\{F\}, \{H, I\}] = \{H, I\}$

root	A	B	C	D	E	F	G	H	I
LIS	$\emptyset$	$\emptyset$	$\emptyset$	$\{D\}$	$\{G\}$	$\{H, I\}$	$\{G\}$	$\{H\}$	$\{I\}$

Similarly, we compute the LIS of nodes B and C.

- $LIS(B) =$Larger$[\{B\} \cup LIS(G), LIS(D) \cup LIS(E)] =$ Larger$[\{B, G\}, \{G, D\}] = \{G, D\}$
- $LIS(C) =$Larger$[\{C\} \cup LIS(H) \cup LIS(I), LIS(F)] =$ Larger$[\{C, H, I\}, \{H, I\}] = \{C, H, I\}$

root	A	B	C	D	E	F	G	H	I
LIS	$\emptyset$	$\{G, D\}$	$\{C, H, I\}$	$\{D\}$	$\{G\}$	$\{H, I\}$	$\{G\}$	$\{H\}$	$\{I\}$

Finally, knowing all the other LIS, we can find the LIS for the whole tree.

- $$\begin{aligned} LIS(A) &= \text{Larger}[\{A\} \cup LIS(D) \cup LIS(E) \cup LIS(F), LIS(B) \cup LIS(C)] \\ &= \text{Larger}[\{A, D, G, H, I\}, \{C, D, G, H, I\}] = \{C, D, G, H, I\} \end{aligned}$$

root	A	B	C	D	E	F	G	H	I
LIS	$\{C, D, G, H, I\}$	$\{G, D\}$	$\{C, H, I\}$	$\{D\}$	$\{G\}$	$\{H, I\}$	$\{G\}$	$\{H\}$	$\{I\}$

Exercise 4

We need to decide whether it is possible to partition a set $A \subseteq \mathbb{Z}$ ($|A| = n$), into two subsets B and $A \setminus B$ such that they have the same sum. Let's denote the sum of set A as sum(A), then it must be true that: sum$(A) =$ sum$(B) +$ sum$(A \setminus B) = k + k = 2k \implies k = \frac{sum(A)}{2}$, as the two sets are disjoint. The decision problem is expressed as a problem of finding whether there exists a subset of A with a sum equal to the target sum k.

Let's express A as an array, without interest in the order of the elements:

$$A = \begin{pmatrix} A_1 & A_2 & \cdots & A_n \end{pmatrix}$$

There exists a subset of $\{A_1, A_2, \cdots A_i\}$ that sums to j if and only if either of the following is true:

- A subset of $\{A_1, A_2, \cdots A_{i-1}\}$ sums to j.
- A subset of $\{A_1, A_2, \cdots A_{i-1}\}$ sums to $j - A_i$. Thus, $sum(B) + A_i = j - A_i + A_i = j$, meaning that $\{A_1, A_2, \cdots A_i\}$ sums to j.

We can establish the following recursive propositional relation for every possible set $\{A_1, A_2, \cdots A_i\}$ and for every possible sum j:

$$D(i,j) = \begin{cases} \text{True}, & \text{if } j = 0 \\ D(i-1,j), & \text{if } j < A_i \\ D(i-1,j) \text{ or } D(i-1,j-A_i), & \text{if } i < n \end{cases}$$

Set A can have at least one negative element, so we can determine the maximum and minimum sums achievable with the elements of A. Then, we define an array D from 1 to (max_sum - min_sum + 1) to cover positions from min sum to max_sum, ensuring a 1-1 mapping. Using $|min\_sum|$ as an offset, we can calculate all possible sums of set A within this range.

Algorithm 3 Partition problem - Find if there is a subset of A whose sum equals to k

procedure CAN-PARTITION(A)
    if $sum(A)$ is odd **then**
        return False ▷ If sum is odd, A can't be partitioned into two equal subsets.
    end if
    $k \leftarrow \lfloor sum(A)/2 \rfloor$
    $max\_sum \leftarrow$ sum of all positive numbers
    $min\_sum \leftarrow$ sum of all negative numbers
    $r \leftarrow |min\_sum|$
    $p \leftarrow max\_sum + r$
    Initialize an array $D[n,p]$ with **False** values
    for $i = 1$ **to** n **do**
        $D[i,r] \leftarrow$ **True**
    end for
    for $i = 1$ **to** n **do**
        for $j = min\_sum$ **to** max_sum **do**
            $D[i,j+r] \leftarrow D[i-1,j+r]$
            if $min\_sum \le j - A_i \le max\_sum$ **then** ▷ Check if the sum with A_i is valid
                $D[i,j+r] \leftarrow D[i-1,j+r]$ **or** $D[i-1,j-A_i+r]$
            end if
        end for
    end for
    return $D[n,k+r]$
end procedure

Exercise 5

Let **A** be a matrix in $\mathbf{R}^{mxn}$:

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix} \in \mathbf{R}^{mxn}$$

(a) The objective of this task is to determine the maximum sum, where each sum is determined by a the manhattan path in the matrix and the elements it contains, beginning from [1, 1] and ending in [m, n]. It is clear that the only way to reach an element $A[i, j]$ in the matrix is through $A[i-1, j]$ or $A[i, j-1]$ (from left or from above).

We can use an array C of size $m \times n$ to calculate the max sum of reaching each element of array A. C(i, j) represents the max sum to reach the element located at coordinates i, j. We can observe the following:

- The sum of reaching the first element (A_{11}) of array A is its own value.
- For elements in the first row (excluding the first element), the max sum of each element is its value plus the corresponding sum of reaching the element directly to the left.
- For elements in the first column (excluding the first element), the max sum of each element is its value plus the corresponding sum of reaching the element directly above.

The recursive relation which solves the problem is the following:

$$C(i,j) = \begin{cases} A_{ij} & \text{, if } i = 1 \text{ and } j = 1 \\ A_{ij} + C(i, j-1) & \text{, if } i = 1 \text{ and } j \geq 2 \\ A_{ij} + C(i-1, j) & \text{, if } j = 1 \text{ and } i \geq 2 \\ max\{C(i-1,j), C(i,j-1)\} + A_{ij} & \text{, otherwise} \end{cases}$$

Algorithm 4 Find Maximum Sum Of the Manhtann Paths from [1, 1] to [m,n]

procedure MAX-SUM-MANHATTAN-DISTANCE(A) $\triangleright$ Let C be an array of size $m \times n$
    $C[1,1] \leftarrow A[1,1]$
    for $i = 2$ **to** m **do**
        $C[i,1] \leftarrow A[i,1] + C[i-1,1]$
    end for
    for $j = 2$ **to** n **do**
        $C[1,j] \leftarrow A[1,j] + C[1,j-1]$
    end for
    for $i = 2$ **to** m **do**
        for $j = 2$ **to** n **do**
            $C[i,j] \leftarrow max\{C[i-1,j], C[i,j-1]\} + A[i,j]$
        end for
    end for
    return $C[m,n]$
end procedure

The complexity of the provided algorithm is $O(n \cdot m)$:

- In line 3, we have an $O(m)$ loop ($m - 1$ steps).
- In line 6, there is another $O(n)$ loop.
- Lines 9 and 10 contain nested loops with complexity $O(m) \cdot O(n) = O(n \cdot m)$.

The final complexity is:

$$\max\{O(m), O(n), O(n \cdot m)\} = O(n \cdot m)$$

(b) Let L(i,j,k) represent the number of paths from the top-left corner to cell (i,j) such that the sum of the elements along the path is exactly k.

The following observations will enable us to formulate the recursive relation that dynamically solves the problem:

- The path from cell (1, 1) to itself is 1 if the target sum S and the value A_{11} are the same.
- There are no paths from cell (1, 1) to itself if the target sum S and the value A_{11} differ. Also, there are no paths from cell (1, 1) to each cell (i, j) where $k < Aij$.
- For cells (i, 1), it is true that the number of paths starting from (1, 1) with target sum S are exactly the same with those who lead to (i-1, 1) with target sum $S - A_{ij}$.
- Similarly, for cells (1, j) the number of paths starting from (1, 1) to (1, j) with target sum S are exactly the same with those who lead to (1, j-1) with target sum $S - A_{ij}$
- Finally, the ways to reach a cell (i, j), i, j $\geq$ 1 with target sum S are the sum of ways to reach (i-1, j) and (i, j-1) with for both target sum $S - A_{ij}$.

All the above can be summarized in recursive relation below:

$$L(i,j,k) = \begin{cases} 1 & \text{, if } (i,j,k) = (1,1,A_{11}) \\ 0 & \text{, if } ((i,j) = (1,1) \land k \neq A_{11}) \lor (k < A_{ij}) \\ L(i-1,j,k-A_{ij}) & \text{, if } i \geq 1 \text{ and } j = 1 \\ L(i,j-1,k-A_{ij}) & \text{, if } i = 1 \text{ and } j \geq 1 \\ L(i-1,j,k-A_{ij}) + L(i,j-1,k-A_{ij}) & \text{, if } i \geq 1 \text{ and } j \geq 1 \end{cases}$$

For this task we will use a similar logic with that in exercise 4.

Algorithm 5 Count the Manhtann Paths from [1, 1] to [m,n] with a turget sum

1: **procedure** COUNT-PATHS-WITH-TARGET-SUM(A, S)
2: $max\_sum \leftarrow$ sum of all positive numbers
3: $min\_sum \leftarrow$ sum of all negative numbers
4: $r \leftarrow |min\_sum|$
5: $p \leftarrow max\_sum + r$
6: Initialize L[i, j, p] with zeros
7: **if** $0 \leq A_{11} \leq p$ **then**
8: $L[1, 1, A_{11} + r] \leftarrow 1$
9: **end if**
10: **for** $i = 1$ **to** n **do**
11: **for** $j = 1$ **to** m **do**
12: **for** $k = min\_sum$ **to** max_sum **do**
13: **if** $min\_sum \leq k - A_{ij} \leq max\_sum$ **then**
14: **if** $i > 1$ **and** $j = 1$ **then**
15: $L[i, j, k + r] = L[i - 1][j][k + r - A_{ij}]$
16: **end if**
17: **if** $j > 1$ **and** $i = 1$ **then**
18: $L[i, j, k + r] = L[i][j - 1][k + r - A_{ij}]$
19: **end if**
20: **if** $j > 1$ **and** $i > 1$ **then**
21: $L[i, j, k + r] = L[i - 1][j][k + r - A_{ij}] + L[i][j - 1][k + r - A_{ij}]$
22: **end if**
23: **end if**
24: **end for**
25: **end for**
26: **end for**
27: **return** $L[m, n, S + r]$
28: **end procedure**

Lines 6,7 and 8 contain nested loops with complexity $O(m) \cdot O(n) \cdot O(S) = O(n \cdot m \cdot S)$. Therefore, the complexity is: $O(n \cdot m \cdot S)$.