ΤΜΗΜΑ
ΗΛΕΚΤΡΟΛΟΓΩΝ
ΜΗΧΑΝΙΚΩΝ &
ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ
ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΑΝΑΛΥΣΗ ΚΑΙ ΣΧΕΔΙΑΣΜΟΣ ΑΛΓΟΡΙΘΜΩΝ

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών ΑΠΘ

Ε εξάμηνο

Θεσσαλονίκη, Απρίλιος 2024

Υπεύθυνοι Εργασίας:

Ονομ/νυμο: Μανούση Βασιλική

ΑΕΜ: 10677

Email: vmanousi@ece.auth.gr

Ονομ/νυμο: Μανωλίδου Ανατολή

ΑΕΜ: 10874

Email: amanolid@ece.auth.gr

Περιεχόμενα

Πρόβλημα 1

Ερώτημα 1ο

Δεδομένου ότι έχουμε ένα μη κατευθυνόμενο γράφο G, αυτός μπορεί να είναι είτε συνδεδεμένος είτε όχι. Στη περίπτωση που δεν είναι συνδεδεμένος, υπάρχει το ενδεχόμενο μεταξύ των πόλεων s και t να μην υπάρχει καν δρομολόγιο, ενώ αν είναι συνδεδεμένος ο γράφος, τότε υπάρχει σίγουρα δρομολόγιο. Παρόλο αυτά, επειδή το αυτοκίνητο μπορεί να εκτελέσει L χιλιόμετρα με γεμάτο ρεζερβουάρ και μπορεί να γεμίσει βενζίνη μόνο μέσα σε μια πόλη υπάρχει περίπτωση ενώ ο γράφος είναι συνδεδεμένος το δρομολόγιο να μην είναι εφικτό. Γι΄ αυτό το λόγο, θα αφαιρέσουμε από τον γράφο όσες ακμές έχουν βάρος $l_e > L$.

Από το μάθημα γνωρίζουμε πως ο αλγόριθμος DFS (Depth First Search) ελέγχει εάν δύο κόμβοι v_i, v_j του γράφου G είναι συνδεδεμένοι. Έτσι θα χρησιμοποιήσουμε αυτό τον αλγόριθμο και πιο συγκεκριμένα θα έχουμε επιπλέον μια λίστα με μέγεθος όσο και το πλήθος των κόμβων του γράφου, στην οποία όλες οι τιμές αρχικά θα είναι false και όταν επισκεπτόμαστε ένα κόμβο η τιμή θα γίνεται true(τα indexes της λίστας θα αντιστοιχούν αλφαβητικά στους κόμβους):

I. Ξεκινάμε από τον κόμβο “s”.
II. Αφού τρέξει ο αλγόριθμος θα θεωρούμε ότι έχει τερματίσει όταν έχουμε επιστρέψει στον κόμβο “s” και δεν υπάρχουν γειτονικοί κόμβοι που να μπορούμε να επισκεφθούμε.
III. Κοιτώντας τώρα τη λίστα που είχαμε δημιουργήσει στην αρχή, εάν στο index όπου αντιστοιχεί ο κόμβος “t” η λίστα επιστρέφει false τότε δεν υπάρχει εφικτό δρομολόγιο εάν όμως επιστρέφει true τότε υπάρχει εφικτό δρομολόγιο μεταξύ των δύο πόλεων.

Ο αλγόριθμος DFS τρέχει σε χρόνο $O(|V| + 2|E|)$, καθώς σε undirected γράφο επισκεπτόμαστε 2 φορές κάθε ακμή. Βέβαια γνωρίζουμε πως $O(|V| + 2|E|) = O(|V| + |E|)$.Ο έλεγχος που προσθέσαμε στην αρχή έχει πολυπλοκότητα $O(|E|)$ (αφού κάνουμε |E| συγκρίσεις και η κάθε σύγκριση έχει $O(1)$), οπότε συνολικά ο αλγόριθμος μας τρέχει σε χρόνο $O(|V| + |E|) + O(|E|) = O(|V| + |E|)$.

Ερώτημα 2ο

Εμείς ψάχνουμε το μικρότερο δυνατό ρεζερβουάρ που μπορεί να έχει το αυτοκίνητο ώστε να ταξιδέψει από την πόλη “s” στη “t”. Άρα, κοιτώντας όλα τα πιθανά μονοπάτια μεταξύ των δυο πόλεων, εάν από το καθένα επιλέξουμε την ακμή με το μεγαλύτερο βάρος(ώστε να μπορεί να κινηθεί το αυτοκίνητο από τον ένα κόμβο στον άλλο) και μετά από όλες αυτές τις τιμές, επιλέξουμε την ακμή με το ελάχιστο βάρος, θα έχουμε απαντήσει στο ερώτημα. Αφού η απάντηση βρίσκεται ανάμεσα στα βάρη των ακμών, θα πάρουμε όλα τα βάρη και με τον αλγόριθμο merge sort θα τα ταξινομήσουμε σε μια λίστα. Έπειτα, θα ξεκινήσουμε ένα binary search στη λίστα και κάθε τιμή που παίρνουμε θα την ορίζουμε ως L’(το νέο μας ρεζερβουάρ δηλαδή) και θα τρέχουμε τον αλγόριθμο που υλοποιήσαμε στο 1ο ερώτημα. Έτσι εάν ο αλγόριθμος δείξει ότι υπάρχει εφικτό δρομολόγιο, θα συνεχίσουμε το binary search στο εύρος τιμών μικρότερου του L’ καθώς υπάρχει περίπτωση να βρούμε μικρότερη τιμή για το L’. Εάν όμως ο αλγόριθμος δεν βρει εφικτό δρομολόγιο θα πρέπει να συνεχίσουμε το binary search στο εύρος τιμών μεγαλύτερου του L’. Η αναζήτηση θα σταματήσει όταν το high index και το low index απέχουν κατά μια μονάδα, όταν έχουν δηλαδή εξερευνηθεί όλες οι δυνατές περιπτώσεις που απαιτούν το μικρότερο βάρος. Στο τέλος θα πρέπει να επιλεχθεί το low εάν η DFS(low) επιστρέφει true ή το high εάν η DFS(high) επιστρέφει true αλλά η DFS(low) επιστρέφει false.

Όσο αφορά τη πολυπλοκότητα του αλγορίθμου μας, από το μάθημα γνωρίζουμε πως οι αλγόριθμοι merge sort και binary search τρέχουν σε χρόνους $O(|E|log(|E|))$ και $O(log(|E|))$ αντίστοιχα. Ο αλγόριθμος του ερωτήματος 1, έχει $O(|V| + |E|)$ και τρέχει τόσες φορές όσες κάνουμε binary search. Άρα ο αλγόριθμος έχει: $O(|E|log(|E|))+O(|V|+|E|)*O(log|E|) = O(|V| + |E| * log|E|)$, επειδή ο όρος $O(|V|+|E|)*O(log|E|)$ επικρατεί του $O(|E|log(|E|))$.Από τη θεωρία των γράφων ξέρουμε πως σε undirected γράφο οι ακμές παίρνουν μια max τιμή ίση με $|E| = ((V * (V - 1))/2)$, άρα $O(log|E|) = O(log(V * (V - 1))/2) = O(log|V^2|) = O(2log|V|) = O(log|V|)$. Τελικά, ο αλγόριθμος έχει το ζητούμενο χρόνο εκτέλεσης δηλαδή, $O(|V|+|E|)*O(log|V|) = O(|V| + |E| * log|V|)$.

Πρόβλημα 2

Ερώτημα 1ο

Εύρεση βέλτιστου αλγορίθμου:

Ζητούμενο του προβλήματος, είναι η ελαχιστοποίηση του συνολικού χρόνου αναμονής n πολιτών σε μια δημόσια υπηρεσία. Αυτό συνεπάγεται ότι θα πρέπει να ελαχιστοποιήσουμε και τον χρόνο αναμονής του εκάστοτε πολίτη. Έτσι θα ορίσουμε μια λίστα με όρισμα n, ενώ $t_1,t_2,...t_n$ οι χρόνοι εξυπηρέτησης των έκαστων πολιτών και T τον συνολικό χρόνο αναμονής των πολιτών. Σύμφωνα με την εκφώνηση, αφού κάθε πολίτης έχει ως χρόνο αναμονής το άθροισμα των χρόνων εξυπηρέτησης των πολιτών που προηγούνται από τον ίδιο, ο συνολικός χρόνος T υπολογίζεται ως εξής:

$$T= 0 + t_1 + (t_1+t_2) + (t_1+t_2+t_3)+ ....+ (t_1+t_2+...+t_{n-1})$$

Παρατηρώντας τον τύπο του T, αντιλαμβανόμαστε πως για να ελαχιστοποιηθεί θα πρέπει οι όροι που επαναλαμβάνονται τις περισσότερες φορές να έχουν και το μικρότερο “βάρος”. Άρα ο χρόνος t_1, ο οποίος επαναλαμβάνεται τις περισσότερες φορές(συγκεκριμένα n-1 φορές) θα πρέπει να είναι ο μικρότερος χρόνος εξυπηρέτησης από όλους . Στη συνέχεια ο t_2 επαναλαμβάνεται μια φορά λιγότερη από τον t_1(δηλαδή n – 2 φορές) άρα επιθυμούμε να έχει την αμέσως μικρότερη τιμή μετά τον t_1. Όλος αυτός ο συλλογισμός καταλήγει στο συμπέρασμα πως για να ελαχιστοποιηθεί ο συνολικός χρόνος εξυπηρέτησης θα πρέπει οι πολίτες να εξυπηρετηθούν με κριτήριο τον χρόνο εξυπηρέτησης τους. Πιο συγκεκριμένα, προτεραιότητα έχουν οι πολίτες με τον μικρότερο χρόνο εξυπηρέτησης. Επιπλέον, απ’ όλα τα παραπάνω, αντιλαμβανόμαστε πως ο συγκεκριμένος αλγόριθμος που επιλέξαμε αποτελεί ένα παράδειγμα άπληστου αλγορίθμου.

Απόδειξη της ορθότητας του:

Έστω ο A και ο B πολίτης με t_A και t_B να είναι οι αντίστοιχοι χρόνοι εξυπηρέτησης τους, και $t_A < t_B$.

- Αν ο A εξυπηρετηθεί πρώτος τότε ο B θα περιμένει τον χρόνο εξυπηρέτησης του A δηλαδή t_A.
- Αν ο B εξυπηρετηθεί πρώτος τότε ο A θα περιμένει τον χρόνο εξυπηρέτησης του B δηλαδή t_B.

Επειδή ισχύει ότι $t_A < t_B$, γίνεται φανερό ότι η κατάταξη των χρόνων εξυπηρέτησης κατά αύξουσα σειρά και η αντίστοιχη εξυπηρέτηση των πολιτών ελαχιστοποιεί τον συνολικό χρόνο αναμονής. Σε διαφορετική περίπτωση, όπου πρώτος εξυπηρετείται ο πολίτης με μεγαλύτερο χρόνο εξυπηρέτησης, ο συνολικός χρόνος αναμονής αυξάνεται.

Πρόβλημα 3

Ερώτημα 1ο

Για να κατανοήσουμε το πρόβλημα καλύτερα, δημιουργήσαμε ένα παράδειγμα. Έστω ότι έχουμε μια συμβολοσειρά μήκους n=6 και θέλουμε να την κόψουμε σε τρία σημεία, τα 1,3,4. Γνωρίζοντας πως πρέπει να προσεγγίσουμε το πρόβλημα με δυναμικό προγραμματισμό επιλέγουμε να ακολουθήσουμε την μέθοδο tabulation, προκειμένου να αποφύγουμε την επανάληψη πράξεων, η οποία οδηγεί σε αύξηση του χρόνου εκτέλεσης. Αυτό σημαίνει πως θα λύνουμε υποπροβλήματα, θα τα αποθηκεύουμε σε ένα πίνακα και θα τα χρησιμοποιούμε για μεγαλύτερα υποπροβλήματα έως ότου να λυθεί όλο το πρόβλημα. Εδώ το κύριο πρόβλημα είναι το ελάχιστο κόστος τομής μεταξύ των σημείων 0 και n = 6 το οποίο αποτελείται από τα εξής υποπροβλήματα:

Κάνοντας τη πρώτη τομή (επιλογή σημείου 1 ή 3 ή 4) η συμβολοσειρά "σπάει" σε δύο τμήματα, ένα αριστερά από το σημείο και ένα δεξιά. Κάθε καινούργιο τμήμα αποτελεί ένα νέο υποπρόβλημα. Παρομοίως γίνεται και η δεύτερη τομή και ούτως καθεξής. Επειδή όμως υπάρχουν πολλοί συνδυασμοί τομών εμείς θα πρέπει να τους υπολογίσουμε όλους και να επιλέξουμε αυτόν με το ελάχιστο κόστος.

Λαμβάνοντας υπόψη όλα τα παραπάνω, δημιουργούμε έναν τετραγωνικό πίνακα "tab[][]", στον οποίο το στοιχείο tab[i][j] θα ορίζεται ως το ελάχιστο κόστος όταν κόβουμε μεταξύ των σημείων i (αρχή) και j(τέλος). Άρα ο πίνακας θα έχει διαστάσεις ίσες με το πλήθος των τομών συν δυο ακόμη τομές, τις 0 και n(δηλαδή την αρχή και το τέλος της συμβολοσειράς έτσι ώστε να είναι δυνατό να υπολογίσουμε όλα τα πιθανά κόστη). Επιπλέον, θα δημιουργήσουμε ένα πίνακα "cuts[]" στον οποίο θα αποθηκεύσουμε τα σημεία τομής καθώς επίσης και τα στοιχεία 0 και n. Έπειτα θα τον ταξινομήσουμε κατά αύξουσα σειρά για την ευκολία των πράξεων. Για δική μας ευκολία ορίζουμε το μέγεθος του cuts[] ως k.

Καταλήγουμε λοιπόν πως τα στοιχεία του πίνακα [1]υπολογίζονται με τον παρακάτω αλγόριθμο:

```
tab[i][j] = min(tab[i][j], cuts[j] – cuts[i] + tab[i + 1][j] + tab[i][i + 1]);
```

Κάθε φορά ο αλγόριθμος συγκρίνει τη τιμή που έχει ήδη η θέση [i][j] του πίνακα(το κόστος δηλαδή ενός από τους συνδυασμούς) με κάποιον από τους υπόλοιπους συνδυασμούς και αποθηκεύει την ελάχιστη, έως να υπολογιστούν όλοι οι συνδυασμοί. Πιο αναλυτικά το κόστος υπολογίζεται από το μήκος της επιμέρους συμβολοσειράς(cuts[j] – cuts[i]) και από το κόστος του αριστερού (tab[i][i + 1])και δεξιού τμήματος (tab[i + 1][j]) που προκύπτουν από τη τομή, όπως είχαμε αναφέρει και παραπάνω. Τελικά η λύση του προβλήματος είναι το στοιχείο tab[0][n - 1] = [0][5] όπου είναι το minimum κόστος για τομή μεταξύ των σημείων 0 και n = 6.

Ερώτημα 2ο

Όσον αφορά το χρόνο εκτέλεσης του αλγορίθμου, βλέποντας και το παράρτημα του προβλήματος 3 έχουμε ως εξής: Το κομμάτι: *tab[i][j] = Math.min(tab[i][j], cuts[j] – cuts[i] + tab[i + 1][j] + tab[i][i + 1])* εκτελείται σε χρόνο $O(1)$. Αυτός ο αλγόριθμος όμως βρίσκεται μέσα σε ένα loop το οποίο τρέχει τόσες φορές όσο και οι πιθανές τομές μεταξύ του i και j, στη χειρότερη περίπτωση θα τρέξει (k – 2) φορές δηλαδή μιλάμε για $O(k-2) = O(k)$. Όλο αυτό το τμήμα βρίσκεται μέσα σε ένα διπλό loop το οποίο τρέχει εξωτερικά από 2 έως k και εσωτερικά από 0 έως k, δηλαδή περίπου σε χρόνο $O(k^2)$. Όλα τα υπόλοιπα μέρη του κώδικα δεν έχουν σχέση με τη λύση του προβλήματος, άρα τελικά ο αλγόριθμος μας τρέχει σε χρόνο $(O(k)*O(k^2)*O(1)) = O(k^3)$.

[1]Βλ. παράρτημα προβλήματος 3 για περαιτέρω λεπτομέρειες

Παραρτήματα

Παρακάτω, για το πρόβλημα 1 παρατίθεται ψευδοκώδικας ενώ για τα προβλήματα 2 και 3 παρατίθεται κώδικας σε Java. Επιπλέον, για τα προβλήματα 2 και 3 παρατίθεται και link από το GitHub στο οποίο είναι αναρτημένοι οι κώδικες για καλύτερη ανάγνωση(τα .java αρχεία βρίσκονται στο φάκελο src). Για το πρόβλημα 3, παρατίθενται και περαιτέρω λεπτομέρειες για τα στάδια του αλγορίθμου.

Πρόβλημα 1

I. Ψευδοκώδικας για το ερώτημα 1:

Σχετικά με τον παρακάτω αλγόριθμο:

- G είναι ο γράφος
- V είναι η λίστα με όλους τους κόμβους του G
- adj είναι ο adjacency πίνακας του G
- visited είναι η λίστα που αποθηκεύουμε τους επισκεπτόμενους κόμβους
- L το ρεζερβουάρ του αυτοκινήτου
- E είναι η λίστα με όλες τις ακμές του G

```
1  Question1(G, V, E, L){
2     for each v ∈ V
3        v.visited = false;
4
5     for each e ∈ E //Removing edges with l_e > L
6        if e.weight > L
7           G.remove[e];
8
9     for each v ∈ V
10       if(v.visited = false) //We can only visit a vertex if it hasn't already been visited
11          DFS(G, s);
12
13    return (t.visited); //Returns true if there is a feasible route between "s" and "t", otherwise returns false
14 }
15
16 DFS(G,v){
17    v.visited = true;
18
19    for each k∈ G.adjacent[v]
20       if k.visited = false
21          DFS(G,k);
22 }
```

Παραρτήματα

II. Ψευδοκώδικας για το ερώτημα 2:

Σχετικά με τον παρακάτω αλγόριθμο:

- Η merge sort δεν αναλύθηκε καθώς δεν χρειάστηκε κάποια τροποποίηση στον αλγόριθμο που έχουμε διδαχτεί στο μάθημα
- W είναι η λίστα με τα βάρη των ακμών
- Question1() είναι συνάρτηση του αλγορίθμου από το ερώτημα 1

```
1  Question2(W){
2     W.mergesort ; //Sorting the weights
3
4     Print("The minimum tank capacity is:  Binarysearch(W));
5  }
6
7  Binarysearch(W){
8        low = 0;
9        high = W.size -1
10
11     while(high.index != low.index + 1){///The search stops when the high index is next to low index
12           L' = (low + high)/2;
13
14           if (Question1(L') returns true){ //If there is a feasible route we set high to L'
15                high = L';
16                go to line 11;
17           }
18           else{ //If there isn't a feasible route we set low to L'
19                low = L';
20                go to line 11;
21           }
22     }
23
24     if(Question1(low) returns true) //The minimum tank capacity is the low value if there is a feasible route
25        return low;
26     else  //The minimum tank capacity is the high value if there isn't a feasible route for the low value
27        return high;
28  }
```

Παραρτήματα

Πρόβλημα 2

https://github.com/AnatoliManolidou/ADA_Problem2

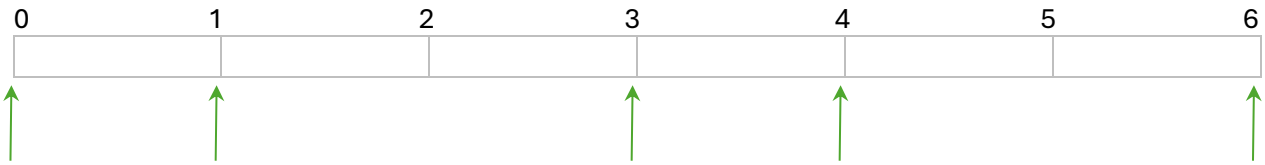
I. *Πρόγραμμα σε Java:*

```
import java.util.Scanner;
import java.util.*;
import java.lang.Math;

public class Problem2 {

    public static void main(String[] args){ //Main method

        Scanner sc = new Scanner(System.in);

        System.out.println("Choose how many citizens are going to be in the
queue"); //Letting the user choose the number of citizens
        int c = sc.nextInt();

        System.out.println("Choose a maximum service time"); //Letting the
user choose a maximum value for the service time
        int max = sc.nextInt();

        List<Integer> list=new ArrayList<Integer>(c); //Creating a list to
store the service time per citizen

        for(int i = 0; i < c; i++) //Filling the list with random values
within the range that the user set previously on line14
            list.add((int)(Math.random() * max));

        Collections.sort(list); //Sorting the list, as our algorithm suggests
        int [] times = new int[c]; //Creating an array to store the waiting
time per citizen

        System.out.println(list); //Printing the list (with each service
time)

        for(int i = 0; i < c; i++)
            times[i] = 0;

        for(int j = 1; j < c; j++) //Every citizen's waiting time is
calculated as the sum of the waiting time from all the citizens ahead of him
            for(int k = j - 1; k >= 0; k --)
                times[j] = list.get(k) + times[j];

        int total = 0;

        for(int j = 0; j < c; j++) //T= 0 + t1 + (t1+t2) + (t1+t2+t3)+ ….+
(t1+t2+...+tn-1)
            total = total + times[j];

        System.out.println("The minimum waiting time is:" + total);
```

Παραρτήματα

Πρόβλημα 3

I. Περαιτέρω λεπτομέρειες:



Εικόνα 1: Ο πίνακας cuts=[0,1,3,4,6] και τα βελάκια τα οποία δείχνουν τα σημεία τομής.

Πίνακας 1:Ο πίνακας tab

	0	1	2	3	5
0	0	0	3	7	12
1	MAX INTEGER	0	0	3	8
2	MAX INTEGER	MAX INTEGER	0	0	3
3	MAX INTEGER	MAX INTEGER	MAX INTEGER	0	0
4	MAX INTEGER	MAX INTEGER	MAX INTEGER	MAX INTEGER	0

***Πως συμπληρώθηκε ο πίνακας?**

Αρχικά γεμίζουμε τον πίνακα σε όλες τις θέσεις με μια MAX INTEGER τιμή (αυτό γίνεται έτσι ώστε στην πρώτη σύγκριση η ελάχιστη τιμή να μπορεί να αποθηκεύεται). Έπειτα, τα στοιχεία της διαγωνίου τα θέτουμε μηδέν καθώς δεν υπάρχει κόστος για τομή με ίδια αρχή και ίδιο τέλος. Επίσης τα στοιχεία του tab[m][m+1] είναι μηδενικά καθώς δεν υπάρχει κόστος μεταξύ δύο διαδοχικών τομών. Ενώ τα στοιχεία κάτω από την διαγώνιο παραμένουν MAX INTEGER καθώς δεν μπορούμε να κόψουμε όταν το j>i. Όσον αφορά τα υπόλοιπα στοιχεία εφαρμόζουμε τον αλγόριθμο που παρουσιάσαμε στη σελ. 5.

Στον υπολογισμό έχουμε δύο περιπτώσεις:

- Στην πρώτη, μεταξύ των i και j υπάρχει μια μόνο δυνατή τομή οπότε γίνεται η σύγκριση του MAX value με το κόστος, δηλαδή το μήκος της επιμέρους συμβολοσειράς.
- Στην δεύτερη, μεταξύ των i και j υπάρχουν παραπάνω από 1 πιθανή τομή οπότε η πρώτη σύγκριση γίνεται όπως στην πρώτη περίπτωση και στη συνέχεια η δεύτερη γίνεται μεταξύ της τιμή που έχουμε ήδη αποθηκεύσει με την επόμενη δυνατή τομή και επικρατεί η ελάχιστη των δύο. Αυτό γίνεται έως να καλυφθούν όλοι οι συνδυασμοί.

Τελικά η λύση του προβλήματος είναι το στοιχείο [0][n - 1] = [0][5] όπου είναι το minimum κόστος.

***Προτεραιότητα πράξεων**

Επειδή ο υπολογισμός κάποιων στοιχείων του πίνακα προϋποθέτει των υπολογισμό κάποιων άλλων, πρέπει να ακολουθήσουμε μια συγκεκριμένη αλληλουχία πράξεων. Όπως αναφέραμε παραπάνω ξεκινάμε πάντα με τον υπολογισμό του στοιχείου [0][2](αφού τα [0][0] και [0][1] είναι πάντα μηδενικά). Έπειτα για να υπολογίσουμε το στοιχείο [0][3] έπρεπε πρώτα να έχει ήδη υπολογιστεί το [1][3] και το [0][4]. Αντίστοιχα για τον υπολογισμό του [0][4] χρειάζεται και το [1][4]. Από όλα τα παραπάνω παρατηρήσαμε ότι προκύπτει ένα μοτίβο όπου πάντα ξεκινάμε από το στοιχείο [0][2] ενώ το επόμενο στοιχείο υπολογισμού θα είναι αυτό της επόμενης σειράς και της επόμενης στήλης. Αυτό γίνεται ως να φτάσουμε στην τελευταία στήλη του πίνακα που επιστρέφουμε στην σειρά 0 και επαναλαμβάνουμε την ίδια διαδικασία.

Παραρτήματα

https://github.com/AnatoliManolidou/ADA_Problem3

II. Πρόγραμμα σε Java:

```
import java.util.Arrays;
import java.util.Scanner;

public class Problem3 {

    public static void main(String[] args) {//Main method

        Scanner sc = new Scanner(System.in);// Scanner object

        System.out.println("Enter the length of the string");
        int n = sc.nextInt();
        System.out.println("Enter the number of cuts that you want to
perform");
        int m = sc.nextInt();
        int[] cuts = new int[m + 2];
        cuts[0] = 0;
        cuts[m + 1] = n;
        System.out.println("Enter the positions of the cuts");
        for(int i = 0; i < m; i++){
            int k = sc.nextInt();
            if(k < n)
                cuts[i] = k;
            else {
                System.out.println("Invalid input, insert new position");
                i--;
            }
        }

        Arrays.sort(cuts); //Sorting so we can compare the different cases

        for (int i = 0; i < cuts.length; i++) { //Validation of the cuts
            int cut = cuts[i];
            System.out.println(cut);
        }

        int mincost =  MinCost(cuts);
        System.out.println("The minimum cost to cut the string in the given
positions is:" + mincost);
    }
        public static int MinCost(int[] cuts) {

            int[][] tab = new int[cuts.length][cuts.length]; // tab comes
from tabulation

            for (int i = 0; i < cuts.length; i++)
```

Παραρτήματα

```
                for (int j = 0; j < cuts.length; j++)
                    tab[i][j] = Integer.MAX_VALUE; // setting the array
values with a large int in order to calculate the min in the following nested
for loop

            //int max_length = (int) (Math.log10(Integer.MAX_VALUE) + 1);

            tab[0][0] = 0; //Standard values in array
            tab[0][1] = 0; //Standard values in array
            tab[cuts.length - 1][cuts.length - 1] = 0; //Standard values in
array

            for (int i = 1; i <= (cuts.length - 2); i++) //Standard values in
array
                for (int j = i; j < (i + 2); j++)
                        tab[i][j] = 0;

            for (int i = 0; i < cuts.length; i++) { //Just printing
                for (int j = 0; j < cuts.length; j++) {
                    System.out.print(tab[i][j]);
                    //for(int l = 0; l <= ((max_length - ((int)
(Math.log10(tab[i][j]) + 1))) + 1); l++){
                        System.out.print(" ");
                    //}
                }
                System.out.println();
                System.out.println();
            }

            System.out.println("          -          ");

            for (int i = 2; i < (cuts.length); i++)  //Tabulation Algorithm
                for (int j = 0; j < (cuts.length - 2); j++) {

                    if(j == 0)
                        for (int x = j + 1; x < i; x++)
                            tab[j][i] = Math.min(tab[j][i],cuts[i] - cuts[j]
+ tab[x][i] + tab[j][x]);
                    else {
                        int h = j + i;
                        if(h < cuts.length)
                            for (int x = j + 1; x < h; x++)
                                tab[j][h] = Math.min(tab[j][h], cuts[h] -
cuts[j] + tab[x][h] + tab[j][x]);
                        }

                        for (int b = 0; b < cuts.length; b++) { //Just printing
                            for (int g = 0; g < cuts.length; g++) {
                                System.out.print(tab[b][g]);
                                System.out.print(" ");
                            }
                            System.out.println();
                            System.out.println();
                        }
                        System.out.println("          -          ");
                }
```

Παραρτήματα

```
            for (int i = 0; i < cuts.length; i++) { //Just printing
                for (int j = 0; j < cuts.length; j++) {
                    System.out.print(tab[i][j]);
                    System.out.print(" ");
                }
                System.out.println();
            }

        return tab[0][cuts.length - 1]; //The final answer of the problem
    }
}
```