Τελική εργασία στο μάθημα Ειδικά Θέματα Αλγορίθμων

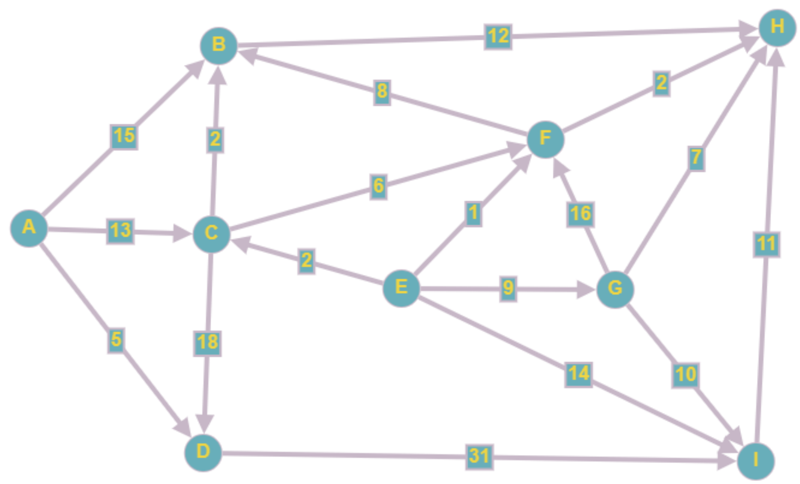
ΠΕΤΡΟΓΙΑΝΝΟΠΟΥΛΟΣ ΓΕΩΡΓΙΟΣ

February 23, 2023

ΑΣΚΗΣΗ 1

Ερώτημα: Α

E- C : 2022201900182 mod 9 = 2
F- H : 2022201900182 mod 3 = 2



Round	Visit	A	B	C	D	F	E	G	H	I	VISITED	UNVISITED
1		-	-	-	-	-	-	-	-	-	[ ]	A,B,C,D,E,F,G,H,I
2	A:		15	13	5	-	-	-	-	-	A	B,C,D,E,F,G,H,I
3	D:		-	-	-	-	-	-	-	31+5=36	A,D	B,C,E,F,G,H,I
4	I:		-	-	-	-	-	-	5+31+11=47	-	A,D,I	B,C,E,F,G,H
5	C:		13+2=15	-	13+18=31	13+6=19	-	-	-	-	A,D,C,I	B,E,F,G,H
6	B:		-	-	-	-	-	-	15+12=27	-	A,D,C,I,B	E,F,G,H
7	F:		13+6+8=27	-	-	-	-	-	13+6+2=21	-	A,D,C,I,B,F	E,G,H
8	E,G,H		-	-		-	-	-	-	-	A,D,C,I,B,F,E,G,H	[ ]
	Min		15	13	5	19	-	-	21	36		
	Paths		A-B	A-C	A-D	A-C-F			A-C-F-H	A-D-I		

Επεξήγηση:

Round 1: Αρχικοποίηση με άπειρο (αντικαταστάθηκε με παύλα) σε όλους τους κόμβους
Round 2: Ξεκινώντας απο τον A κοιτάμε τους γείτονες , και σημειώνουμε τα κόστη τους. Αφού γίνει αυτό , σημειώνουμε visited το A και βγαινει απο unvisited.
Round 3: Τώρα επιλέγουμε τον γείτονα του A με το μικρότερο κόστος απο τους unvisited , αυτός είναι ο D. Αρα πλεον για τον D βλέπουμε οτι έχει μονο εναν γείτονα τον I και σημειώνουμε το κόστος του. D visited , και βγαινει απο unvisited.
Round 4: Τώρα πάμε στον I , επιλέγουμε τον γείτονα του I με το μικρότερο κόστος απο τους unvisited , αυτός είναι ο H με κόστος (A-D)+(D-I)+(I-H)=5+31+11=47. I visited , και βγαινει απο unvisited.
Round 5: Τώρα επιλέγουμε τον γείτονα του A με το μικρότερο κόστος απο τους unvisited, αμέσως μικρότερο κόστος έχει ο C. Για τον C πάμε στο B όμως αυτή τη φορά έχουμε το κόστος (A-C) + (C-B) = 13+2= 15. Μετά πάμε στον D , με

κόστος (A-C) + (C-D) = 13+18 =31. Τέλος πάμε στον F με κόστος (A-C) + (C-F) = 13+6=19, και σημειώνουμε F στα visited και βγαίνει απο unvisited

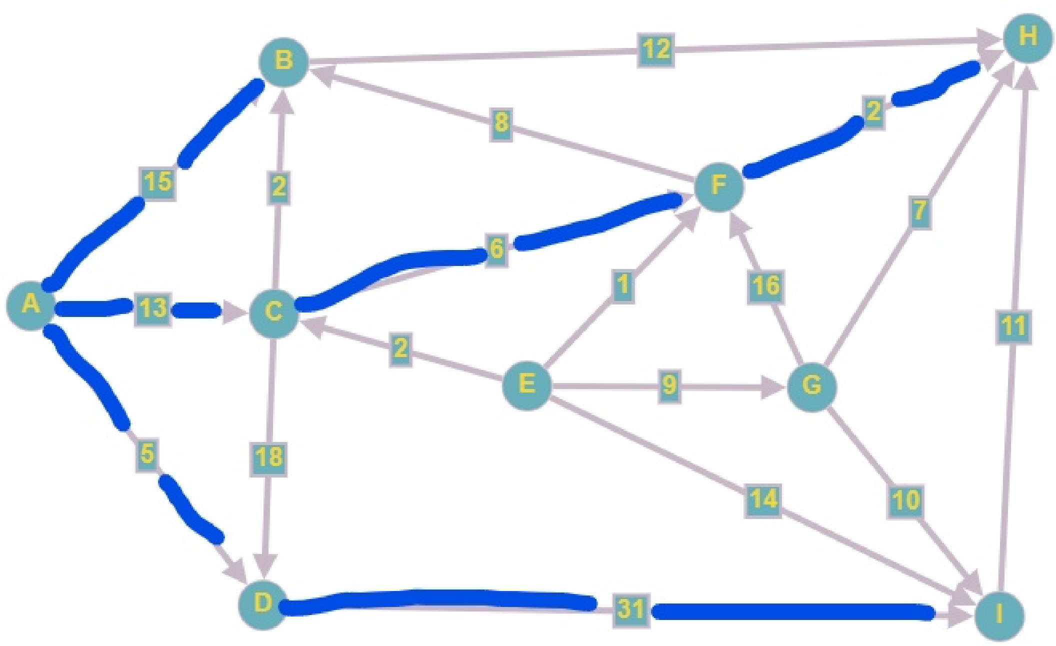
Round 6: Επιλέγουμε τον γείτονα του A με το μικρότερο κόστος απο τους unvisited ,τον B , για τον B πάμε μόνο στον H με κόστος (A-B) + (B-H)=15+12=27, σημειώνουμε B visited και βγαίνει απο unvisited

Round 7: Τώρα κοιτάμε τους γείτονες του C που είναι unvisited , αυτός είναι μονο ο F , για τον F μπορούμε να πάμε σε B με κόστος:(A-C)+(C-F)+(F-B)=13+6+8=27, και στον H με κόστος (A-C)+(C-F)+(F-H)=13+6+2=21, και βάζουμε τον F visited και βγαίνει απο unvisited

Round 8: ο H δεν έχει εξερχόμενες ακμές , και οι E,G δεν συνδέονται ούτε άμεσα ούτε έμμεσα με τον A , συνεπώς ο αλγόριθμος δεν ψάχνει για αυτούς.

Τελικά:, απο κάθε στήλη για κάθε κόμβο επιλέγουμε το μικρότερο κόστος , για παράδειγμα στην στήλη του κόμβου B , επιλέφουμε το 15 καθώς είναι μικρότερο απο το 27, και το path είναι A-B. Για τον C είναι 13, path A-C

Για τον D είναι 5 αφού 5<31, και path A-D Για τον F είναι 19 , path A-F Για τον E,G κενό. Για τον H είναι 21 αφού 21<27<47, path A-C-F-H Για τον I είναι 36, path A-D-I. Δηλαδή το τελικό αποτέλεσμα είναι η γραμμή min του πίνακα, και τα paths τα οποία μαζί συνθέτουν το παρακάτω shortest path:



Ερώτημα: B

Ενα παράδειγμα είναι , αν είχαμε κόστος -31 απο D $- >$ I , αυτό θα επηρέαζε το κόστος απο το A μέχρι το H και είναι λάθος διότι πλεον θα έδειχνε ως μικρότερο μονοπάτι το A-D-I-H και οχι το A-C-F-H καθώς ψάχνει "άπληστα" πάντα τη μικρότερη διαδρομή. Για να εκτελεστεί σωστά θα πρέπει υποχρεωτικά να χρησιμοποιήσουμε αλλον αλγόριθμο , τον Bellman-Ford , ο οποίος μπορεί να υπολογίζει ελάχιστα μονοπάτια και για γράφους με αρνητικές ακμές.

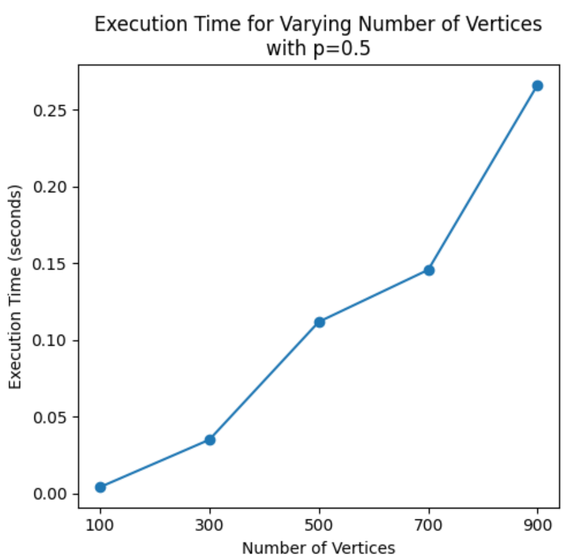
Ερώτημα: Γ

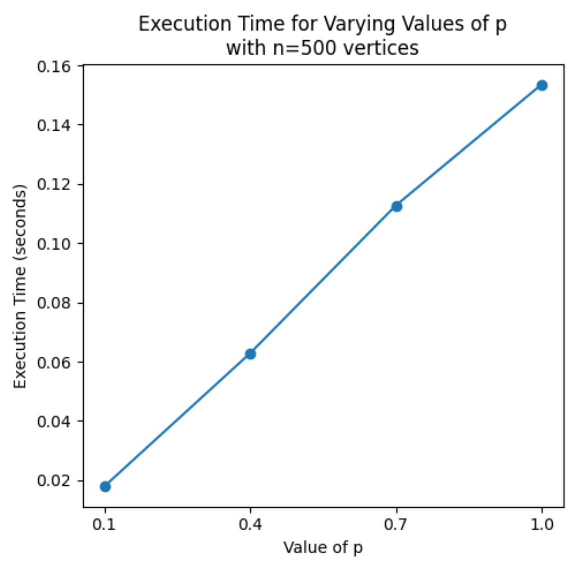
```python
import networkx as nx #pip install networkx
import random
import heapq
import time

def dijkstra(G, source):
    dist = {v: float('inf') for v in G.nodes()}
    prev = {v: None for v in G.nodes()}
    dist[source] = 0
    q = [(0, source)]
    while q:
        u_dist, u = heapq.heappop(q)
        if u_dist > dist[u]:
            continue
        for v in G.neighbors(u):
            alt = dist[u] + G[u][v]['weight']
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                heapq.heappush(q, (dist[v], v))
    return dist, prev

# From 100 to 900 vertices with p=0.5
p = 0.5
increments = [100, 300, 500, 700, 900]
for n in increments:
    G = nx.erdos_renyi_graph(n, p)
    for u, v in G.edges():
        G[u][v]['weight'] = random.randint(1, 10)
    source = 0
    start_time = time.time()
    dist, prev = dijkstra(G, source)
    end_time = time.time()
    print(f"{n} vertices with p = {p}: {end_time - start_time:.5f} seconds")

# 500 vertices with p from 0.1 to 1 with increment of 0.3
```

```python
increments = [0.1, 0.4, 0.7, 1.0]
for p in increments:
    n = 500
    G = nx.erdos_renyi_graph(n, p)
    for u, v in G.edges():
        G[u][v]['weight'] = random.randint(1, 10)
    source = 0
    start_time = time.time()
    dist, prev = dijkstra(G, source)
    end_time = time.time()
    print(f"{n} vertices with p = {p}: {end_time - start_time:.5f} seconds")
```

Αποτελέσματα:

```
100 vertices with p = 0.5: 0.00399 seconds
300 vertices with p = 0.5: 0.03491 seconds
500 vertices with p = 0.5: 0.11173 seconds
700 vertices with p = 0.5: 0.14564 seconds
900 vertices with p = 0.5: 0.26630 seconds
500 vertices with p = 0.1: 0.01790 seconds
500 vertices with p = 0.4: 0.06283 seconds
500 vertices with p = 0.7: 0.11270 seconds
500 vertices with p = 1.0: 0.15356 seconds
```





Παρατηρώ τα εξής:
Για ίδιο p , όσο αυξάνονται οι κορυφές τόσο περισσότερος χρόνος χρειάζεται αλλά

δεν αυξάνεται ραγδαία , δηλαδή όχι με πολύ μεγάλο ρυθμό, ενώ για το ανάποδο , δηλαδή για σταθερές κορυφές και αυξανόμενο p όσο αυξάνεται το p αυξάνεται πάλι ο χρόνος εκτέλεσης αλλα με μεγαλύτερη αύξηση/μεγαλύτερο ρυθμό. Συνεπώς τα αποτελέσματα με σταθερό p δίνουν χαμηλότερο execution time επειδή έχουν μικρότερο ρυθμό αύξησης.

ΑΣΚΗΣΗ 2

Ερώτημα: Α

```
Δεν λύνει σωστά το πρόβλημα διότι:
Για i=1:
    h_2 +1 > l_1+l_2
      51  > 11  (True)
        εβδομάδα 1 ->    = 0
        εβδομάδα 2 -> h  = 50
        continue to i+2->3
Για i=3:
    h_4 +1 > l_3+l_4
      2  > 20  (False)
    else
      εβδομάδα 3 -> l   = 10
      continue to i+1->4
Για i=4:
    h_5 +1 > l_4+l_5
      ERROR, δεν υπάρχει h_5 και l_5
Αρα θα βγάλει κόστος 0+50+10=60 ,
που δεν είναι βέλτιστο καθώς υπάρχει το βέλτιστο που
είναι 70 στο παράδειγμα της εκφώνησης.

Παραδειγμα:
    Εβδομάδα 1 Εβδομάδα 2 Εβδομάδα 3 Εβδομάδα 4
low    11        10          1        13
high    5        50          5        1

Σωστή απάντηση:
    εβδομάδα 1 = low = 11
    εβδομάδα 2 = high = 50
    εβδομάδα 3 = 0
    εβδομάδα 4 = low  = 13
    σύνολο = 74

Απάντηση ψευδοκώδικα(μαζί με τον περιορισμό υπερχείλισης):
    εβδομάδα 1 = low = 5
    εβδομάδα 2 = low = 50
    εβδομάδα 3 = 0
    εβδομάδα 4 = high  = 13
    σύνολο = 68 Αρα είναι πάλι λάθος
```

Ερώτημα: Β

Ο παραπάνω αλγόριθμος χρησιμοποιεί δυναμικό προγραμματισμό για να βρει το καλύτερο πλάνο για high και low jobs. Ξεκινά δημιουργώντας έναν πίνακα maxjob μεγέθους n+1, όπου κάθε ευρετήριο αντιπροσωπεύει μια ημέρα. Ο αλγόριθμος γεμίζει τον πίνακα επαναλαμβάνοντας κάθε μέρα (από την ημέρα 2 έως την ημέρα ν) και για κάθε ημέρα, υπολογίζει τις μέγιστες εργασίες που μπορούν να γίνουν μέχρι εκείνη την ημέρα.
Ο αλγόριθμος έχει την εξής λογική,πως κάθε μέρα υπάρχουν δύο επιλογές - είτε να κάνετε μια εργασία high είτε μια εργασία low. Έτσι, για κάθε ημέρα i, απαιτούνται το μέγιστο δύο περιπτώσεις: είτε κάνουμε μια εργασία high την ημέρα i και τις μέγιστες εργασίες που έγιναν μέχρι την ημέρα i-2, είτε κάνουμε μια εργασία low την ημέρα i και τις μέγιστες εργασίες μέχρι την ημέρα i-1:

```
max_job[i] = max(h[i - 1] + max_job[i - 2],l[i - 1] + max_job[i - 1] )
```

Τέλος, η συνάρτηση επιστρέφει τις μέγιστες εργασίες που μπορούν να γίνουν μέχρι την ημέρα n, που είναι maxjob[n].
Η πολυπλοκότητα αυτού του αλγορίθμου είναι O(n) καθώς χρησιμοποιεί δυναμικό προγραμματισμό για να αποφύγει υπολογισμούς. Υπολογίζει τις μέγιστες εργασίες για κάθε ημέρα i χρησιμοποιώντας τις μέγιστες εργασίες που υπολογίζονται για τις ημέρες i-1 και i-2. Συνεπώς ο υπολογισμός κάθε ημέρας γίνεται μόνο μία φορά και ο αλγόριθμος δεν επαναλαμβάνει τον ίδιο υπολογισμό πολλές φορές.

Ερώτημα: Γ

Αποδοτικός(Dynamic programming)

```
# Returns maximum amount of task,that can be done till day n
def maxTasks(h, l, n):
        # An array max_job that stores,he maximum task done
        max_job = [0] * (n + 1);
        # If n = 0, no solution exists
        max_job[0] = 0;
        # If n = 1, high effort task,solution is here
        max_job[1] = h[0];
        # Fill the entire array with a task to choose on each day i
        for i in range(2, n + 1):
                max_job[i] = max( h[i - 1] + max_job[i - 2],l[i - 1] + max_job[i - 1] );

        return max_job[n];

n=4
h = [11, 10, 1, 13]
l = [5, 50, 5, 1]
print(maxTasks(h, l, n));
```

Heuristic, first fit

```
def first_fit(low, high, n): # First fit heuristic
    # Initialize the schedules
    low_schedule = []
    high_schedule = []

    # Iterate over the tasks
    for i in range(n):
        # Check if the task can be assigned to a low cost worker
        if low[i] <= (50 - sum(low_schedule)):
            low_schedule.append(low[i])
        # Otherwise, check if the task can be assigned to a high cost worker
        elif high[i] <= (130 - sum(high_schedule)):
            high_schedule.append(high[i])
        # If neither is possible, assign the task to a high cost worker
        else:
            high_schedule.append(high[i])

    # Return the total number of tasks assigned
    return len(low_schedule) + len(high_schedule)

n=4
h = [11, 10, 1, 13]
l = [5, 50, 5, 1]
print(first_fit(h, l,n));
```

Testing

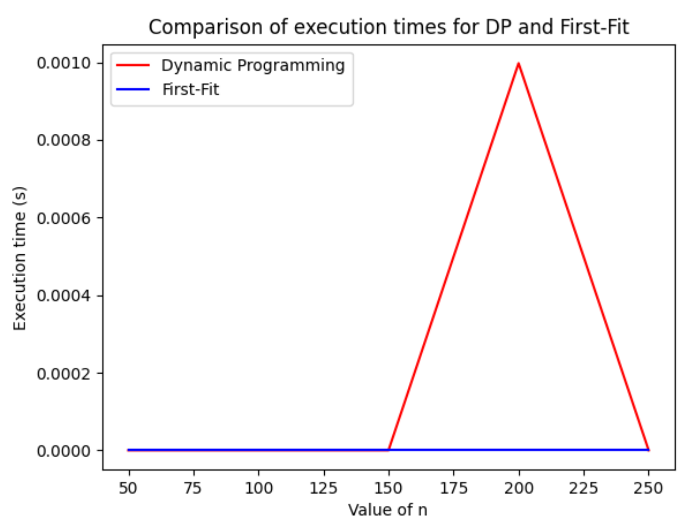
```
import random
import time
from  matplotlib import pyplot as plt
# Returns maximum amount of task,that can be done till day n
def maxTasks(h, l, n):
    max_job = [0] * (n + 1)
    max_job[0] = 0
    max_job[1] = h[0]
    for i in range(2, n + 1):
        max_job[i] = max(h[i - 1] + max_job[i - 2], l[i - 1] + max_job[i - 1])

    return max_job[n]

def first_fit(low, high, n): # First fit heuristic
    low_schedule = []
    high_schedule = []

    for i in range(n): # traverse the tasks
        # Check if the job can be assigned to a low effort job
        if low[i] <= (50 - sum(low_schedule)):
            low_schedule.append(low[i])
        # Check if the job can be assigned to a high effort job
        elif high[i] <= (130 - sum(high_schedule)):
            high_schedule.append(high[i])
        else: # assign the job to a high cost effort job
            high_schedule.append(high[i])

    # total number of tasks assigned
    return len(low_schedule) + len(high_schedule)


# Driver code
n_values = [50, 100, 150, 200, 250]
dp_times = []
ff_times = []
for n in n_values:
    # Generate random task costs
    low = [random.randint(0, 50) for i in range(n)]
    high = [random.randint(90, 130) for i in range(n)]

    # Time the dynamic programming algorithm
    dp_start_time = time.time()
    dp_schedule = maxTasks(high, low, n)
    dp_end_time = time.time()
```

```python
    dp_times.append(dp_end_time - dp_start_time)

    # Time the first fit heuristic
    heuristic_start_time = time.time()
    heuristic_schedule = first_fit(low, high, n)
    heuristic_end_time = time.time()
    ff_times.append(heuristic_end_time - heuristic_start_time)

    # Print the results
    print(f"n={n}")
    print(f"Dynamic Programming: {dp_schedule} tasks,
        {dp_end_time-dp_start_time:.5f} seconds")
    print(f"First Fit Heuristic: {heuristic_schedule} tasks,
        {heuristic_end_time-heuristic_start_time:.5f} seconds\n")

# Plot the execution times for both algorithms
plt.plot(n_values, dp_times, 'r-', label='Dynamic Programming')
plt.plot(n_values, ff_times, 'b-', label='First-Fit')

# Set the x and y labels
plt.xlabel('Value of n')
plt.ylabel('Execution time (s)')

# Set the title and legend
plt.title('Comparison of execution times for DP and First-Fit')
plt.legend(loc='best')

# Show the plot
plt.show()
```

Αποτελέσματα:

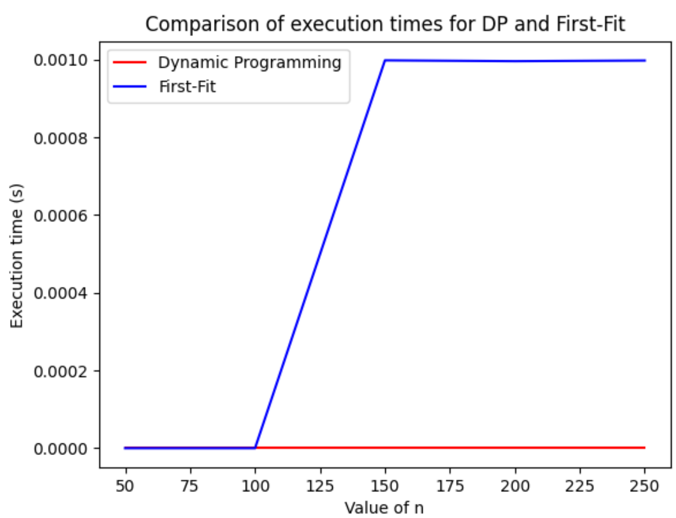
Δοκιμή 1

```
n=50
Dynamic Programming: 2879 tasks,0.00000 seconds
First Fit Heuristic: 50 tasks, 0.00000 seconds

n=100
Dynamic Programming: 5682 tasks,0.00000 seconds
First Fit Heuristic: 100 tasks, 0.00000 seconds

n=150
Dynamic Programming: 8524 tasks,0.00000 seconds
First Fit Heuristic: 150 tasks, 0.00000 seconds

n=200
Dynamic Programming: 11281 tasks,0.00100 seconds
First Fit Heuristic: 200 tasks, 0.00000 seconds

n=250
Dynamic Programming: 13889 tasks,0.00000 seconds
First Fit Heuristic: 250 tasks, 0.00000 seconds
```



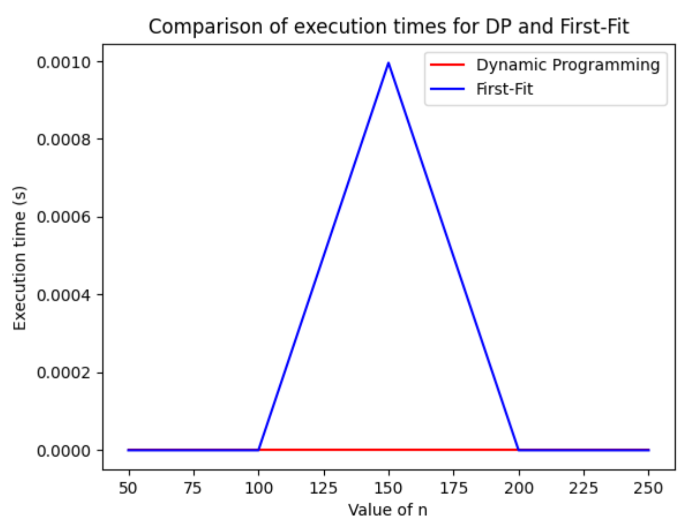
Δοκιμή 2

```
n=50
Dynamic Programming: 2939 tasks,0.00000 seconds
First Fit Heuristic: 50 tasks, 0.00000 seconds

n=100
Dynamic Programming: 5677 tasks,0.00000 seconds
First Fit Heuristic: 100 tasks, 0.00000 seconds

n=150
Dynamic Programming: 8514 tasks,0.00000 seconds
First Fit Heuristic: 150 tasks, 0.00100 seconds

n=200
Dynamic Programming: 11166 tasks,0.00000 seconds
First Fit Heuristic: 200 tasks, 0.00100 seconds

n=250
Dynamic Programming: 14080 tasks,0.00000 seconds
First Fit Heuristic: 250 tasks, 0.00100 seconds
```



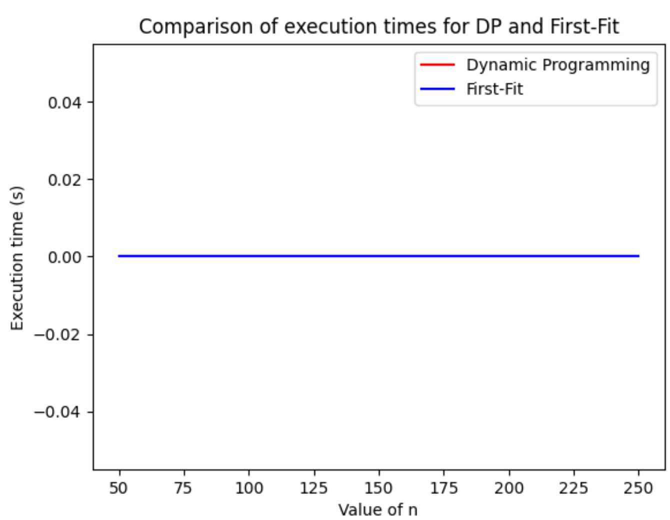
Δοκιμή 3

```
n=50
Dynamic Programming: 2777 tasks,0.00000 seconds
First Fit Heuristic: 50 tasks, 0.00000 seconds

n=100
Dynamic Programming: 5640 tasks,0.00000 seconds
First Fit Heuristic: 100 tasks, 0.00000 seconds

n=150
Dynamic Programming: 8513 tasks,0.00000 seconds
First Fit Heuristic: 150 tasks, 0.00100 seconds

n=200
Dynamic Programming: 11063 tasks,0.00000 seconds
First Fit Heuristic: 200 tasks, 0.00000 seconds

n=250
Dynamic Programming: 14038 tasks,0.00000 seconds
First Fit Heuristic: 250 tasks, 0.00000 seconds
```



Δοκιμή 4

```
n=50
Dynamic Programming: 2874 tasks,0.00000 seconds
First Fit Heuristic: 50 tasks, 0.00000 seconds

n=100
Dynamic Programming: 5682 tasks,0.00000 seconds
First Fit Heuristic: 100 tasks, 0.00000 seconds

n=150
Dynamic Programming: 8462 tasks,0.00000 seconds
First Fit Heuristic: 150 tasks, 0.00000 seconds

n=200
Dynamic Programming: 11378 tasks,0.00000 seconds
First Fit Heuristic: 200 tasks, 0.00000 seconds

n=250
Dynamic Programming: 13995 tasks,0.00000 seconds
First Fit Heuristic: 250 tasks, 0.00000 seconds
```



ΑΣΚΗΣΗ 3

Ερώτημα: Α

Συνολα:

$F = 1, 2, \ldots, n$:σύνολο εργοστασίων
$W = 1, 2, \ldots, n$: σύνολο αποθηκών
$S = 1, 2, \ldots, n$: σύνολο καταστημάτων

Παράμετροι:

m: αριθμός παραγωγής προιόντων εργοστασίων , χωρητικότητας αποθηκών , και απαίτησης προιόντων καταστημάτων

c_{ij}: το κόστος μεταφοράς μιας μονάδας προιόντος απο εργοστάσιο i στην αποθήκη j

d_{jk}: το κόστος μεταφοράς μιας μονάδας προιόντος απο αποθήκη j στο κατάστημα k

Μεταβλητές:

$x_{ijk} \in 0, 1$: binary μεταβλητή που δείχνει αν μια μονάδα μεταφέρεται απο το εργοστάσιο i στην αποθήκη j και μετά στο κατάστημα k

Objective function:

$\min_{x_{ijk}} \sum_{i \in F} \sum_{j \in W} \sum_{k \in S} (c_{ij} + d_{jk}) x_{ijk}$

Περιορισμοί:

$\sum_{j \in W} x_{ij1} = m \quad \forall i \in F$
$\sum_{k \in S} x_{ink} = \sum_{j \in W} x_{ijn} \quad \forall i \in F$
$\sum_{i \in F} x_{ijk} \leq m \quad \forall j \in W, k \in S$
$x_{ijk} \in 0, 1 \quad \forall i \in F, j \in W, k \in S$

Ο **πρώτος** περιορισμός διασφαλίζει ότι κάθε εργοστάσιο παράγει ακριβώς m μονάδες προϊόντος.
Ο **δεύτερος** περιορισμός διασφαλίζει ότι η συνολική ποσότητα του προϊόντος που μεταφέρεται από το εργοστάσιο i στην αποθήκη n ισούται με τη συνολική ποσότητα του προϊόντος που μεταφέρεται από την αποθήκη n στο κατάστημα k.
Ο **τρίτος** περιορισμός διασφαλίζει ότι το η συνολική ποσότητα προϊόντος που μεταφέρεται από κάθε αποθήκη δεν υπερβαίνει τη δική της χωρητικότητα m μονάδων.
Ο **τέταρτος** είναι ότι οι μεταβλητές απόφασης είναι δυαδικές μεταβλητές που πάρτε την τιμή 1 εάν και μόνο εάν μια μονάδα προϊόντος μεταφέρεται από το εργοστάσιο i στην αποθήκη j και στη συνέχεια στο κατάστημα k. Η objective function ελαχιστοποιεί το σύνολο κόστος μεταφοράς

Ερώτημα: Β

Ένας αποδοτικός αλγόριθμος για την επίλυση αυτού του προβλήματος γραμμικού προγραμματισμού θα ήταν να βρίσκει το ελάχιστο κόστος μεταφοράς μεταξύ των n,εργοστασίων ,n αποθηκών και n καταστημάτων με μια πολυπλοκότητα που θα εξαρτάται απο τον αριθμο των εργοστασίων , αποθηκών και καταστημάτων. Για παράδειγμα μια πολυπλοκότητα της μορφής: $\mathrm{O}(n^{(\alpha\rho\iota\theta\mu o\varsigma\epsilon\rho\gamma o\sigma\tau\alpha\sigma\omega\nu,\alpha\pi o\theta\eta\kappa\nu\kappa\alpha\iota\kappa\alpha\tau\alpha\sigma\tau\eta\mu\tau\omega\nu)})$
Πιο συγκεκριμένα , ο αλγόριθμος πρέπει αρχικα να ορίσει τις μεταβλητές απόφασης , δηλ τον αριθμό των εργοστασίων , αποθηκών και καταστημάτων και τον αριθμό των προιόντων που μεταφέρονται κάθε φορά. Ύστερα πρέπει να οριστεί η objective function ως το άθροισμα απο όλα τα κόστη μεταφοράς απο κάθε εργοστάσιο σε κάθε αποθήκη , και από κάθε αποθήκη σε κάθε κατάστημα.
Μετά , ορίζονται οι περιορισμοί, και λύνει το πρόβλημα.
Περαιτέρω πληροφορίες αναφορικά με τον αλγόριθμο ύπαρχουν στο ερώτημα Γ.

Ψευδοκώδικας:

```
n = 3  # Number of factories, warehouses, and stores
m = 5  # number of products
factories,warehouses,stores = n

generate random transportion costs for:
    from factory i to warehouse j:
    from warehouse j to store k

define decision variables
    create dictionairy x that includes them

Define the objective function
    sum of all costs for every factory to warehouse to store

Define constraints
    for each warehouse : check if x[warehouse] <= m
    for each store : check if x[store] == m

solve problem
print optimal solution steps and total minimum cost
```

Ερώτημα: Γ

Κώδικας , παρακάτω η επεξήγηση του τρόπου λειτουργιάς του, και τέλος οι δοκιμές:

```python
from pulp import * # pip install pulp
import random      # included in python , no need to install
# complexity O(n^(Number of factories, warehouses, and stores)), in this case n=3 so O(n^3)

# Define the problem as a minimization problem
prob = LpProblem("Transhipment Problem", LpMinimize)

# Define the decision variable (n),
n = 3 # Number of factories, warehouses, and stores
m = 5 # number of products
factories = range(n)
warehouses = range(n)
stores = range(n)

# Define the transportation costs
# Transportation cost from factory i to warehouse j, random number from 1 to 10
f_w = {(i,j): random.randint(1,10) for i in factories for j in warehouses}
# Transportation cost from warehouse j to store k, random number from 1 to 10
w_s = {(j,k): random.randint(1,10) for j in warehouses for k in stores}

# Define the decision variables
x = LpVariable.dicts("x", ((i,j,k) for i in factories for j in warehouses for k in stores),

# Define the objective function
prob += lpSum(f_w[i,j] * x[i,j,k] for i in factories for j in warehouses for k in stores) \
        + lpSum(w_s[j,k] * x[i,j,k] for i in factories for j in warehouses for k in stores)

# Define the constraints
# Capacity constraint for each warehouse
for j in warehouses:
    prob += lpSum(x[i,j,k] for i in factories for k in stores) <= m

# Demand constraint for each store
for k in stores:
    prob += lpSum(x[i,j,k] for i in factories for j in warehouses) == m

# Solve the problem
prob.solve()

print("|- n = ",n,", m = ",m)
print("|----------- Costs ------------")
print("+ Factories  -> Warehouses:  f_w(F,W): cost")
```

```python
print("|-> f_w = ",f_w)
print("+ Warehouses -> Stores:      w_s(W,S): cost")
print("|-> w_s = ",w_s)
print("|------------------------------")

# Print the results
print("+------ Optimal Solution ------")
for i in factories:
    for j in warehouses:
        for k in stores:
            if x[i,j,k].value() != 0:
                print(f"|-> Transfer {x[i,j,k].value()} products from factory {i+1}
                to warehouse {j+1} with cost {f_w[i,j]}, and then to store {k+1} with cost
                {w_s[i,j]} (total cost: {f_w[i,j]*x[i,j,k].value()+w_s[j,k]*x[i,j,k].value(
                # explanation/debug
                print(f"|     f_w[{i},{j}] * x[{i},{j},{k}].value() + w_s[{j},{k}] * x[{i},
                print("|        ",f_w[i,j],"   *     ",x[i,j,k].value(),"        +   ",w_s[j
print(f"|= Total minimum cost: {value(prob.objective)}")
```

Περιγραφή λειτουργίας του κώδικα:

Ο κώδικας χρησιμοποιεί τη βιβλιοθήκη PuLP, η οποία είναι βασισμένη σε Python για προβλήματα γραμμικής βελτιστοποίησης. Η βιβλιοθήκη PuLP επιτρέπει στο χρήστη να ορίσει το πρόβλημα με όρους μεταβλητών απόφασης, objective function και περιορισμών και στη συνέχεια λύνει το πρόβλημα χρησιμοποιώντας τον αλγόριθμο simplex ή άλλες τεχνικές βελτιστοποίησης.

Πιο συγκεκριμένα , η ανάλυση του τρόπου λειτουργίας του κώδικα βήμα προς βήμα:

Εισάγει τη βιβλιοθήκη PuLP και την τυχαία βιβλιοθήκη για τη δημιουργία τυχαίων αριθμών (import random).

Ορίζει το πρόβλημα ως πρόβλημα ελαχιστοποίησης χρησιμοποιώντας τη συνάρτηση LpProblem από το PuLP.

Ορίζει αρχικα τα n = 3,m = 5 και τις μεταβλητές απόφασης σε ενα dictionairy $x[i, j, k]$ χρησιμοποιώντας τη συνάρτηση LpVariable από το PuLP. Οι μεταβλητές απόφασης αντιπροσωπεύουν τον αριθμό των προϊόντων που θα μεταφερθούν από κάθε εργοστάσιο i σε κάθε αποθήκη j σε κάθε κατάστημα k

Ορίζει την objective function ως το άθροισμα του κόστους μεταφοράς από κάθε εργοστάσιο σε κάθε αποθήκη ,

`(αντικατέστησα c με f_w και d με w_s για καλύτερη ανάγνωση απο τον κώδικα)`

`f_w[i,j]`

και από κάθε αποθήκη σε κάθε κατάστημα

```
w_s[j,k]
```

, χρησιμοποιώντας τη συνάρτηση lpSum από το PuLP.

Ορίζει τους περιορισμούς του προβλήματος, οι οποίοι περιλαμβάνουν περιορισμούς χωρητικότητας για κάθε αποθήκη και περιορισμούς ζήτησης για κάθε κατάστημα. Οι περιορισμοί προστίθενται στο πρόβλημα χρησιμοποιώντας τον τελεστή += ψάχνοντας μέσα στο dictionairy x που περιέχει τις μεταβλητές απόφασης.

Καλεί τη μέθοδο solve() στο αντικείμενο του προβλήματος για να λύσει το πρόβλημα χρησιμοποιώντας τον αλγόριθμο simplex.

Εκτυπώνει τα αποτελέσματα της βελτιστοποίησης, συμπεριλαμβανομένης της βέλτιστης λύσης δηλαδή του συνολικού ελάχιστου κόστους.

Η χρονική πολυπλοκότητα του κώδικα είναι $O(n^3)$, όπου n είναι ο αριθμός των εργοστασίων, των αποθηκών και των καταστημάτων, συνεπώς εαν είχαμε 2 εργοστάσια,2 αποθήκες και 2 καταστήματα θα ήταν $O(n^2)$. Αυτό συμβαίνει επειδή οι λειτουργίες του κώδικα, όπως η δημιουργία των μεταβλητών απόφασης και των dictionairies κόστους μεταφοράς, και ο καθορισμός της objective function, χρειάζονται όλες χρόνο $O(n^3)$. Ο χρόνος που απαιτείται για την επίλυση του προβλήματος χρησιμοποιώντας τον αλγόριθμο simplex είναι συνήθως πολύ μικρότερος από το $O(n^3)$ και εξαρτάται από το μέγεθος του προβλήματος και τον συγκεκριμένο αλγόριθμο που χρησιμοποιείται από το PuLP.

Δοκιμές με διαφορετικές τιμές εισόδου:

Δοκιμή 1: (n = 2, m = 5, τυχαία κόστη)

```
|- n =  2 , m =  5
|----------- Costs ------------
+ Factories  -> Warehouses:  f_w(F,W): cost
|-> f_w =  {
                (0, 0): 6, (0, 1): 2,
                (1, 0): 9, (1, 1): 7        }
+ Warehouses -> Stores:      w_s(W,S): cost
|-> w_s =  {
                (0, 0): 6, (0, 1): 1,
                (1, 0): 7, (1, 1): 2        }
|-----------------------------
+------ Optimal Solution ------
|-> Transfer 5.0 products from factory 1 to warehouse 1 with cost 6,
    and then to store 1 with cost 6 (total cost: 60.0)
|     f_w[0,0] * x[0,0,0].value() + w_s[0,0] * x[0,0,0].value()
|         6    *      5.0         +    6     *      5.0          =   60.0
|-> Transfer 5.0 products from factory 1 to warehouse 2 with cost 2,
    and then to store 2 with cost 1 (total cost: 20.0)
|     f_w[0,1] * x[0,1,1].value() + w_s[1,1] * x[0,1,1].value()
|         2    *      5.0         +    2     *      5.0          =   20.0
|= Total minimum cost: 80.0
Optimal objective 80 - 0 iterations time 0.002, Presolve 0.00
Problem MODEL has 4 rows, 8 columns and 16 elements
Total time (CPU seconds):       0.02
```

Δοκιμή 2: (n = 2, m = 5, τυχαία κόστη)

```
|- n =  2 , m =  5
|----------- Costs ------------
+ Factories  -> Warehouses:  f_w(F,W): cost
|-> f_w =  {(0, 0): 5, (0, 1): 5, (1, 0): 10, (1, 1): 5}
+ Warehouses -> Stores:      w_s(W,S): cost
|-> w_s =  {(0, 0): 1, (0, 1): 5, (1, 0): 2, (1, 1): 6}
|------------------------------
+------ Optimal Solution ------
|-> Transfer 5.0 products from factory 1 to warehouse 1 with cost 5,
    and then to store 1 with cost 1 (total cost: 30.0)
|     f_w[0,0] * x[0,0,0].value() + w_s[0,0] * x[0,0,0].value()
|         5    *      5.0         +    1     *      5.0          =   30.0
|-> Transfer 5.0 products from factory 1 to warehouse 2 with cost 5,
    and then to store 2 with cost 5 (total cost: 55.0)
|     f_w[0,1] * x[0,1,1].value() + w_s[1,1] * x[0,1,1].value()
|         5    *      5.0         +    6     *      5.0          =   55.0
|= Total minimum cost: 85.0
Optimal objective 85 - 0 iterations time 0.012, Presolve 0.01
Problem MODEL has 4 rows, 8 columns and 16 elements
Total time (CPU seconds):       0.04   (Wallclock seconds):       0.04
```

Δοκιμή 3: (n = 3, m = 5, τυχαία κόστη)

```
|- n =  3 , m =  5
|----------- Costs ------------
Factories  -> Warehouses:  f_w(F,W): cost
f_w =  {    (0, 0): 7, (0, 1): 7, (0, 2): 2,
            (1, 0): 10, (1, 1): 7, (1, 2): 4,
            (2, 0): 6, (2, 1): 3, (2, 2): 1     }
Warehouses -> Stores:      w_s(W,S): cost
w_s =  {    (0, 0): 9, (0, 1): 4, (0, 2): 10,
            (1, 0): 3, (1, 1): 8, (1, 2): 9,
            (2, 0): 7, (2, 1): 10, (2, 2): 2    }
|------------------------------
+------ Optimal Solution ------
|-> Transfer 5.0 products from factory 3 to warehouse 1 with cost 6,
    and then to store 2 with cost 7 (total cost: 50.0)
|     f_w[2,0] * x[2,0,1].value() + w_s[0,1] * x[2,0,1].value()
|         6    *      5.0         +    4     *      5.0           =   50.0
|-> Transfer 5.0 products from factory 3 to warehouse 2 with cost 3,
    and then to store 1 with cost 10 (total cost: 30.0)
|     f_w[2,1] * x[2,1,0].value() + w_s[1,0] * x[2,1,0].value()
|         3    *      5.0         +    3     *      5.0           =   30.0
|-> Transfer 5.0 products from factory 3 to warehouse 3 with cost 1,
    and then to store 3 with cost 2 (total cost: 15.0)
|     f_w[2,2] * x[2,2,2].value() + w_s[2,2] * x[2,2,2].value()
|         1    *      5.0         +    2     *      5.0           =   15.0
|= Total minimum cost: 95.0
Optimal objective 95 - 3 iterations time 0.002, Presolve 0.00
Problem MODEL has 6 rows, 27 columns and 54 elements
Total time (CPU seconds):       0.02
```

Δοκιμή 4: (n = 3, m = 5, τυχαία κόστη)

```
|- n =  3 , m =  5
|----------- Costs ------------
Factories  -> Warehouses:  f_w(F,W): cost
f_w =  {    (0, 0): 4, (0, 1): 5, (0, 2): 5,
            (1, 0): 10, (1, 1): 3, (1, 2): 5,
            (2, 0): 9, (2, 1): 7, (2, 2): 6     }
Warehouses -> Stores:      w_s(W,S): cost
w_s =  {    (0, 0): 2, (0, 1): 5, (0, 2): 7,
            (1, 0): 6, (1, 1): 3, (1, 2): 6,
            (2, 0): 4, (2, 1): 4, (2, 2): 7     }
+------ Optimal Solution ------
|-> Transfer 5.0 products from factory 1 to warehouse 1 with cost 4,
    and then to store 1 with cost 2 (total cost: 30.0)
|     f_w[0,0] * x[0,0,0].value() + w_s[0,0] * x[0,0,0].value()
|         4    *      5.0         +    2     *      5.0          =   30.0
|-> Transfer 5.0 products from factory 1 to warehouse 3 with cost 5,
    and then to store 2 with cost 7 (total cost: 45.0)
|     f_w[0,2] * x[0,2,1].value() + w_s[2,1] * x[0,2,1].value()
|         5    *      5.0         +    4     *      5.0          =   45.0
|-> Transfer 5.0 products from factory 2 to warehouse 2 with cost 3,
    and then to store 3 with cost 3 (total cost: 45.0)
|     f_w[1,1] * x[1,1,2].value() + w_s[1,2] * x[1,1,2].value()
|         3    *      5.0         +    6     *      5.0          =   45.0
|= Total minimum cost: 120.0
Optimal objective 120 - 5 iterations time 0.002, Presolve 0.00
Problem MODEL has 6 rows, 27 columns and 54 elements
Total time (CPU seconds):       0.02
```

Μέχρι στιγμής παρατηρώ οτι ο χρόνος εκτέλεσης δεν αλλάζει για μικρά n , το m και τα τυχαια κόστη πρέπει να σημειωθεί οτι δεν επηρεάζουν την πολυπλοκότητα (για αυτο έχω βάλει 2 δοκιμές για κάθε n ώστε να φανεί οτι δεν επηρεάζει). Στην επόμενη σελίδα δοκιμάζω για μεγαλύτερα n.

εδω επειδή για μεγάλα n τα αποτελέσματα εκτυπώνουν πολλές γραμμές παραθέτω μόνο τα τελικά αποτελέσματά τους:
Δοκιμή 5: (n = 30, m = 5, τυχαία κόστη)

```
Optimal objective 340 - 83 iterations time 0.032, Presolve 0.03
Problem MODEL has 60 rows, 27000 columns and 54000 elements
Total time (CPU seconds):       0.17
```

Δοκιμή 6: (n = 60, m = 5, τυχαία κόστη)

```
Optimal objective 605 - 168 iterations time 0.202, Presolve 0.19
Problem MODEL has 120 rows, 216000 columns and 432000 elements
Total time (CPU seconds):       1.14
```

Συμπερασματικά , παρατηρώ οτι όσο μεγαλύτερο n έχω, τόσο αυξάνεται ο χρόνος εκτέλεσης δηλ το Total time (CPU seconds), το οποίο είναι σωστό αν αναλογιστούμε την πολυπλοκότητα του κώδικα.