Αλγόριθμοι και Πολυπλοκότητα

2η Σειρά Γραπτών Ασκήσεων

Ιωάννης Τσαντήλας, 03120883

Άσκηση 1

Κρατάμε σε έναν πίνακα dp[i][j] το ελάχιστο πλήθος ημερών για να βάψουμε το διάστημα [i, j]. Θέτουμε όλες τις θέσεις ίσες με ∞, ενώ για i=j, dp=1 (αφού θέλουμε μία μέρα για να βάψουμε ένα διάστημα μήκους 1). Εάν $c_i \neq c_j$, εξετάζουμε αναδρομικά όλα τα πιθανά «κοψίματα» [i, k] και [k, j], με $i \leq k < j$ και παίρνουμε το ελάχιστο. Εάν $c_i = c_j$, τότε είναι πιθανό οι πλάκες να βαφτούν την ίδια ημέρα, δηλαδή dp[i+1][j-1]+1. Η λύση θα βρίσκεται στο dp[0][n-1]. Συνολικά:

$$dp[i][j] = min \begin{cases} 1, & if\ i = j \\ \min\limits_{i \leq k < j}(dp[i][k] + dp[k+1][j]), & if\ c_i \neq c_j \\ dp[i+1][j-1] + 1, & if\ c_i = c_j \end{cases}$$

Εφόσον το πεζοδρόμιο αποτελείται από n πλάκες, υπάρχουν n^2 δυνατά διαστήματα να εξετάσουμε, ενώ για κάθε από αυτά θα πρέπει να εξετάσουμε k=n «κοψίματα», δηλαδή η **χρονική πολυπλοκότητα** είναι $O(n^3)$. Η **χωρική πολυπλοκότητα** καθορίζεται από τον πίνακα dp, ο οποίος είναι n×n, άρα $O(n^2)$.

Άσκηση 2

Ερώτημα 1

Ο πυρήνας C ενός prefix του pt σημαίνει πως το C είναι τόσο στην αρχή όσο και στο τέλος του prefix. Εάν C=p, αυτό σημαίνει ότι το p εμφανίζεται στο τέλος αυτού του prefix και άρα μέσα στο ίδιο το t. Έτσι, έχουμε ένα pattern match του p μέσα στο t. Πιο απλά, αν υπάρχει pattern match του p μέσα στο t, τότε το t μπορεί να γραφεί ως t_1-p-t_2. Έτσι, ένα prefix του pt = p-t_1-p-t_2 είναι το p-t_1-p, το οποίο έχει ως πυρήνα το p. Εξετάζοντας όλα τα πιθανά prefix του pt, καλύπτουμε όλα τα πιθανά ενδεχόμενα.

Ερώτημα 2

Αρχικά, πάντα θα υπάρχει ένας k-πυρήνας, αφού στην χειρότερη περίπτωση θα είναι η ϵ. Επομένως, από πλευρά «ύπαρξης» είναι καλά ορισμένη έννοια. Έστω πως πυρήνας του p είναι C, δηλαδή C είναι ο μέγιστος k-πυρήνας, $\text{length}(C) = k_{max}$. Για οποιοδήποτε $k \leq k_{max}$, το αντίστοιχο substring μήκους k του C θα είναι επίσης πυρήνας του p. Εάν $k > k_{max}$ τότε, απλά, θα είναι η ϵ.

Για να βρούμε εάν ένα string p (μήκους l) έχει k-πυρήνα, θα έχουμε έναν μετρητή (έστω c), όπου για c=0 έως k κρατάμε τις θέσεις [0,c] και [l-c, l] του string. Εάν $p_{0,c} = p_{l-c,l}$ τότε κρατάμε την τιμή του c και προχωράμε, κρατώντας στο τέλος την μεγαλύτερη. Μπορούμε να κάνουμε binary search στο c, επιτυγχάνοντας πολυπλοκότητα k·logk. Αυτό, σε python:

```
low = 0
up = l
mid = (up+low)//2
kmax = 0
while(low <= up):
    if(str[:mid] == str[-mid:]):
        kmax = mid
        low = mid
        mid = (up+low)//2
    else:
        up = mid
        mid = (up+low)//2
```

Άσκηση 3

Ερώτημα 1

Για τη τετριμμένη λύση, όπου οι s,t συνδέονται με ακμή, μηδενίζουμε αυτή. Διαφορετικά, θα τρέξουμε έναν Dijkstra 2 φορές, μία στο κανονικό γράφημα με αρχή το s και μία, αντιστρέφοντας τη φορά των ακμών, από τον t. Στην συνέχεια, θα μηδενίσουμε κάθε ακμή (u,v) και θα υπολογίσουμε τη νέα συνολική απόσταση ως:

$$distance[s \to u] + distance[v \to t]$$

Όπου ο 1ος όρος δίνεται από τον 1ο Dijkstra και ο 2ος όρος από τον 2ο Dijkstra. Τέλος, επιλέγουμε τη συντομότερη.

Η πρακτική είναι σωστή, αφού αναγκαστικά θα πρέπει να ελέγξουμε όλες τις δυνατές επιλογές για να αποφασίσουμε τη νέα συντομότερη διαδρομή (μπορεί να είχαμε βέλτιστη διαδρομή 1-2-3 και μία άλλη διαδρομή 1-100, επομένως μηδενίζοντας την 100 προκύπτει η νέα βέλτιστη). Η **χρονική πολυπλοκότητα** του Dijkstra με χρήση binary heap είναι O(m·logn), η αντιστροφή των ακμών για τον 2ο χρειάζεται O(m), ενώ για να ελέγξουμε τον μηδενισμό όλων των ακμών θέλουμε m επαναλήψεις. Συνολικά, O(2(m·logn)+ 2·m) = **O(2·m + 2·m·logn)**.

Ερώτημα 2

Εδώ, θα χρειαστεί να χρησιμοποιήσουμε δυναμικό προγραμματισμό. Πάλι, θα τρέξουμε τον Dijkstra κανονικά και θα αποθηκεύσουμε τις αποστάσεις s-v στον distance[v]. Θα δημιουργήσουμε έναν πίνακα dp[i][v], ο οποίος δηλώνει την ελάχιστη απόσταση του s με τη v εάν i ακμές έχουν μηδενιστεί. Η αρχική συνθήκη είναι πως:

$$dp[0][v] = distance[v]$$

Ενώ θέτουμε όλες τις υπόλοιπες θέσεις dp[i][v] ίσες με άπειρο. Για κάθε i από 1 έως και k, θα γεμίσουμε τον πίνακα ως:

$$dp[i][v] = min\begin{cases} dp[i][v] \\ dp[i][u] + weight(u,v) \\ dp[i-1][u] \end{cases}$$

Που σημαίνει, είτε να μην συμπεριλάβουμε την κορυφή u, να τη συμπεριλάβουμε χωρίς να μηδενίσουμε την ακμή (u,v) και να τη συμπεριλάβουμε μηδενίζοντας την ακμή (u,v). Ο Dijkstra χρειάζεται O(m·log n), ενώ για να γεμίσουμε τον k·n πίνακα θέλουμε m επαναλήψεις, αφού πρέπει να υπολογίσουμε για κάθε πιθανή τιμή του k εάν θα μηδενίσουμε κάθε ακμή ή όχι. Συνολικά, **O(k·m·n + m·log n)**.

Άσκηση 4

Ερώτημα 1

Αφού έχουμε μονοπάτι, έχουμε μόνο δυνατή διαδρομή από το s στο t. Αναδρομικά, έστω πως είμαστε στην πόλη V.

Εάν $V \to t = l > B$, ελέγχουμε τις πόλεις U ($V \to U = \text{distance} \leq B$) και εντοπίζουμε αυτή με τη χαμηλότερη δυνατή τιμή, έστω X.

- Για $cost_V > cost_X$:
  - Εάν το ντεπόζιτο μας επιτρέπει να φτάσουμε στη X τότε πάμε στην X και επαναλαμβάνουμε.
  - Εάν το ντεπόζιτο δεν μας επιτρέπει να φτάσουμε στη X, τότε γεμίζουμε στην V όσο χρειαζόμαστε για να φτάσουμε στην X και επαναλαμβάνουμε.
- Εάν $cost_V < cost_X$ τότε γεμίζουμε το ντεπόζιτο «τέρμα» στην V και μετά πάμε στη X.

Εάν $V \to t = \text{distance} \leq B$, ελέγχουμε τις πόλεις στην διαδρομή $V \to t$. Εάν το $cost_V$ είναι το μικρότερο δυνατό, γεμίζουμε όσο χρειαζόμαστε για να φτάσουμε στην t. Διαφορετικά, γεμίζουμε τόσο όσο να φτάσουμε στην πόλη με το χαμηλότερο κόστος και γεμίζουμε εκεί όσο χρειαζόμαστε για να φτάσουμε στην t. Στην χειρότερη, για κάθε πόλη, θα πρέπει να ελέγξουμε όλες τις επόμενες για ανεφοδιασμό, δηλαδή $(n-1)+(n-2)+\ldots = \frac{n(n-1)}{2}$. Άρα έχουμε **χρονική πολυπλοκότητα** $O(n^2)$.

Ερώτημα 2

Δημιουργούμε έναν πίνακα dp[v][f] που συμβολίζει το ελάχιστο κόστος της διαδρομής s-v με f λίτρα στο ντεπόζιτο, με dp[s][0]=0, ενώ όλες οι υπόλοιπες θέσεις ισούται με ∞. Για κάθε ακμή e=(v,u) και για κάθε επίπεδο ντεπόζιτου f, με $f+b(v,u) \leq B$:

$$dp[v][f] = min\begin{cases} dp[v][f] \\ \min\limits_{u,v} dp[u][f + b(v,u)] + c(u) \cdot b(v,u) \end{cases}$$

Ο 2ος όρος δηλώνει το κόστος να φτάσουμε στην v από την u με f λίτρα (άρα στην u θα έχουμε f+b(v,u). Για να αποφύγουμε κύκλο, θα εκτελέσουμε μία τροποποιημένη DFS.

Κρατάμε σε έναν bool πίνακα visited[n] εάν έχουμε επισκεφτεί την v τουλάχιστον μία φορά και σε μία στοίβα path το μονοπάτι των πόλεων που εξετάζουμε. Για κάθε v με visited[v]=0, εκτελούμε DFS – θέτοντας visited[v]=1 και προσθέτοντας την v στη (κενή) στοίβα. Για κάθε γειτονικό κόμβο u, εάν visited[u]=0, εκτελούμε DFS για αυτόν (με ξεχωριστή στοίβα path), ενώ εάν είναι στην στοίβα, σημαίνει πως βρήκαμε κύκλο και σταματάμε την DFS.

Η λύση είναι η ελάχιστη τιμή της σειράς dp[t][f].

Για να τρέξουμε την DFS για όλους τους κόμβους, θέλουμε $n \cdot (n+m)$ επαναλήψεις, ενώ για να γεμίσουμε τον $n \cdot B$ πίνακα θέλουμε m επαναλήψεις για κάθε ένα από τα B επίπεδα της βενζίνης. Συνολικά, **η χρονική πολυπλοκότητα** είναι $O(n \cdot B^2 \cdot m + n \cdot (n+m)) \sim O(n \cdot B^2 \cdot m + n^2) \sim O(n \cdot B^2 \cdot m)$.

Άσκηση 5

Ερώτημα (α)

Έχουμε ένα διμερές γράφημα Γ = (Κ, Ι, Α), με Κ το σύνολο των κάστρων-κόμβων, Ι το σύνολο των ιπποτών-κόμβων και Α το σύνολο των ακμών, μεταξύ ενός κάστρου k και ενός ιππότη i, εάν τα στρατεύματα του k συμφωνούν να εποπτεύονται από τον ιππότη i (δηλαδή ο i είναι στον κατάλογο K_k). Κάθε ιππότης i μπορεί να επιβλέπει το πολύ c_i κάστρα.

Για να λύσουμε το πρόβλημα, θα δημιουργήσουμε κατάλληλα ένα flow network, με έναν source κόμβο S και έναν sink κόμβο Τ. Συνδέουμε στον S όλα τα κάστρα Κ με βάρος ακμής 1, θέτουμε το βάρος των ακμών Α επίσης ίσο με 1 και συνδέουμε όλους τους ιππότες Ι με τον Τ με βάρος ακμής το αντίστοιχο c_i. Στην συνέχεια, βρίσκουμε το maximum flow (π.χ. με Edmonds-Karp) και ελέγχουμε εάν όλα τα κάστρα εμπεριέχονται στη λύση. Εάν ναι, βρήκαμε το βέλτιστο ταίριασμα, διαφορετικά δεν υπάρχει.

Όσον αφορά τις πολυπλοκότητες, έστω πως V το συνολικό πλήθος των κόμβων και Ε το συνολικό πλήθος των ακμών. Οι κόμβοι θα είναι οι Κ, συν τους Ι, συν 2 για τους S και Τ. Οι ακμές θα είναι Κ (μία για κάθε κάστρο με τον S), συν Ι (μία για κάθε ιππότη με τον Τ), συν το άθροισμα όλων των c_i. Στην χειρότερη περίπτωση, όλοι οι ιππότες συνδέονται με όλα τα κάστρα, επομένως:

$$V = |K| + |I| + 2$$

$$E = |K| + |I| + |K| \cdot |I|$$

Ο Edmonds-Karp έχει χρονική πολυπλοκότητα $O(V \cdot E^2)$ και χωρική $O(V+E)$, δηλαδή:

$$\boldsymbol{Time\ Complexity} = O[(K + I + 2) \cdot (K + I + KI)^2] \sim O(K^2 I^2)$$

$$\boldsymbol{Space\ Complexity} = O(K + I + 2 + K + I + KI) \sim O(KI)$$

Ερώτημα (β)

Ο αλγόριθμος βρίσκει επανειλημμένα επαυξητικά μονοπάτια στο υπολειπόμενο γράφημα και μόλις βρει ένα, αυξάνει τη ροή κατά μήκος αυτού του κατά το μέγιστο δυνατό ποσό, το οποίο καθορίζεται από τη χωρητικότητα των ακμών του. Μετά την αύξηση, ο αλγόριθμος ενημερώνει το υπολειπόμενο γράφημα για να λάβει υπόψη του τη νέα χωρητικότητα: μειώνει τις χωρητικότητες των μπροστά ακμών και αυξάνει τις χωρητικότητες των πίσω ακμών. Ο αλγόριθμος επαναλαμβάνεται μέχρι να μην μπορεί να βρεθεί κανένα επαυξητικό μονοπάτι στο υπολειμματικό γράφημα.

Υπάρχουν κάποιοι βασικοί λόγοι για τους οποίους είμαστε σίγουροι πως θα τερματίσει. Στο γράφημα, κάποια ακμή θα έχει τη μεγαλύτερη (πεπερασμένη) χωρητικότητα, επομένως η τιμή της ροής θα αυξηθεί μέχρι εκείνη τη χωρητικότητα και θα τερματίσει. Επιπλέον, όταν προσπαθεί να βρει νέο επαυξητικό μονοπάτι, χρησιμοποιεί BFS, η οποία δεν μπορεί να επισκεφτεί ξανά κόμβους που έχουν ήδη επισκεφτεί – δηλαδή ο αλγόριθμος εξερευνά όλα τα πιθανά επαυξητικά μονοπάτια (τα οποία είναι πεπερασμένα εν σύνολο αφού έχουμε πεπερασμένες ακμές).

Ερώτημα (γ)

Προσθέτουμε στο αρχικό μας γράφημα μία ακμή ανάμεσα στους κόμβους των ιπποτών για τους οποίους υπάρχει σύγκρουση και σκοπεύουμε να βρούμε το minimum vertex cover, δηλαδή να βρούμε το ελάχιστο πλήθος ιπποτών, που αν αφαιρεθούν, θα εξαφανιστούν όλες οι ακμές σύγκρουσης. Δυστυχώς όμως, το minimum vertex cover είναι NP-hard για μεγάλο μέγεθος εισόδου (για μικρό, υπάρχει ο αλγόριθμος Branch and Bound).

Ωστόσο, υπάρχει ένα ενδεχόμενο να λυθεί αποδοτικά, εάν το καινούργιο γράφημα είναι επίσης διμερές, δηλαδή το σύνολο των ιπποτών μπορεί να χωριστεί σε δύο επιμέρους, όπου κάθε ένας έχει σύγκρουση με ιππότη του άλλου συνόλου, και όχι του ίδιου συνόλου. Σε αυτή την περίπτωση, μπορούμε να αξιοποιήσουμε το θεώρημα του König, δηλαδή το μέγεθος του maximum matching (#ακμών) ισούται με το μέγεθος του minimum vertex cover (#κορυφών).

Εδώ, πράγματι, το νέο γράφημα (τα δύο σύνολα των ιπποτών) είναι διμερές! Επομένως, αρκεί να βρούμε το νέο μέγιστο ταίριασμα, χρησιμοποιώντας τον αλγόριθμο των Hopcroft-Karp. Ο αλγόριθμος θέτει σε κάθε

κορυφή μία τιμή (επίπεδο) που είναι η απόσταση της από την πηγή s και ψάχνει τα συντομότερα μονοπάτια επαύξησης. Μόλις βρεθούν, «αντιστρέφει» τις ακμές – οι αντιστοιχισμένες γίνονται μη αντιστοιχισμένες και αντίστροφα. Ο αλγόριθμος επαναλαμβάνεται μέχρι να μην μπορεί να βρει άλλα μονοπάτια. Η πολυπλοκότητα του είναι $O(\sqrt{V} + E)$.

Στην συνέχεια μπορούμε να εφαρμόσουμε το θεώρημα του König. Ξεκινούμε με όλες τις unmatched κορυφές και ακολουθούμε τα εναλλασσόμενα μονοπάτια από αυτές, προσθέτοντας στο cover:

- Κάθε κορυφή που επισκεπτόμαστε στην unmatched πλευρά μέσω μιας matched ακμής.
- Κάθε κορυφή που φτάνετε στην matched πλευρά που ταιριάζει μέσω μιας unmatched ακμής.

Άσκηση 6

Αρχικά, θα ταξινομήσουμε τις προσφορές με βάση την χρονική στιγμή αποδέσμευσης τους t_i, έτσι ώστε να ξεκινήσουμε με αυτή που έχει μικρότερο τέλος – π.χ. με quicksort. Για κάθε προσφορά i, θα προσπαθήσουμε να βρούμε την επόμενη, πιο πρόσφατη, προσφορά που δεν συγκρούεται χρονικά με την i - π.χ. με binary search. Δημιουργούμε έναν πίνακα revenue[i], που αντιπροσωπεύει το μέγιστο κέρδος από τις i πρώτες προσφορές, με revenue[0] = 0, αφού για 0 προσφορές δεν έχουμε κέρδος. Για κάθε i ϵ [1, n], το revenue[i] θα είναι το μέγιστο μεταξύ:

- Το κέρδος εάν απορρίψουμε την προσφορά i, το οποίο ισούται με το προηγούμενο κέρδος, revenue[i-1].
- Το κέρδος εάν δεχτούμε την προσφορά i, όπου προσθέτουμε στο κέρδος revenue[w] την αξία της προσφοράς, p_i (με revenue[w] να είναι το συνολικό κέρδος έως και την τελευταία μη συγκρουόμενη προσφορά w).

Έχουμε διαθέσιμα k αυτοκίνητα, επομένως θα επεκτείνουμε τον πίνακα μας, revenue[i][j], με i να αντιπροσωπεύει τις προσφορές και j ϵ [0, k] τα αυτοκίνητα. Θέτουμε revenue[0][j] = 0 και revenue[i][0] = 0, αφού εάν δεν έχουμε προσφορές ;ή διαθέσιμα αυτοκίνητα, δεν θα έχουμε κέρδος. Για κάθε προσφορά i, με j διαθέσιμα αυτοκίνητα, το κέρδος revenue[i][j] θα είναι το μέγιστο μεταξύ:

- Το κέρδος εάν απορρίψουμε την προσφορά i, το οποίο ισούται με το προηγούμενο κέρδος, revenue[i-1][j].
- Το κέρδος εάν δεχτούμε την προσφορά i, όπου προσθέτουμε στο κέρδος revenue[w][j-1] την αξία της προσφοράς, p_i (με revenue[w][j-1] να είναι το συνολικό κέρδος έως και την τελευταία μη συγκρουόμενη προσφορά w, με ένα διαθέσιμο αμάξι λιγότερο).

Έτσι, το μέγιστο έσοδο θα βρίσκεται στο revenue[n][k]. Για να δούμε ποιες προσφορές συμμετέχουν, ανατρέχουμε από το revenue[n][k] και ελέγχουμε εάν η προσθήκη μίας προσφοράς αύξησε το κέρδος, στην οποία περίπτωση συμμετέχει στη βέλτιστη λύση. Η αναδρομή σχέση που περιγράφει την επίλυση είναι:

$$revenue(i, j) = \max\left[revenue(i-1, j), p_i + revenue(w(i), j-1)\right]$$

Η συνάρτηση w(i) επιστρέφει τον δείκτη της τελευταίας μη συγκρουόμενης προσφοράς με την i-οστή προσφορά. Εάν δεν υπάρχει επιστρέφει 0. Τέλος, η **πολυπλοκότητα χώρου** είναι το μέγεθος του πίνακα revenue, δηλαδή $\mathbf{O(n \cdot k)}$. Η χρονική πολυπλοκότητα είναι το άθροισμα των:

- Sorting τις προσφορές με βάση τις χρονικές στιγμές t_i, π.χ. με quicksort, $O(n \cdot logn)$.
- Εύρεση της πιο πρόσφατης μη συγκρουόμενης προσφοράς με την i (συνάρτηση w(i)). Αφού είναι ταξινομημένες, με binary search μπορούμε να τη βρούμε σε logn, και αφού έχουμε n προσφορές, $O(n \cdot logn)$.
- Συμπλήρωση του πίνακα, όπου κάθε κελί χρειάζεται O(1) χρόνο, και αφού έχουμε n·k κελιά, θέλουμε $O(n \cdot k)$.

Συνολικά, η **χρονική πολυπλοκότητα** είναι $O(n \cdot k + 2 \cdot n \cdot logn)$ και θεωρώντας τα n, k αρκετά μεγάλα, $\mathbf{O(n \cdot k)}$.